SKS INCLUDED HYPERTEXT DATABASE AND UTILITIES DISKS INCLUDED

COVERS DOS 5.0

DOS UNDOCUMENTED

A PROGRAMMER'S GUIDE TO RESERVED MS-DOS® FUNCTIONS AND DATA STRUCTURES

ANDREW SCHULMAN, RAYMOND J. MICHELS, JIM KYLE, TIM PATERSON, DAVID MAXEY, AND RALF BROWN

UNDOCUMENTED DOS

UNDOCUMENTED DOS

A PROGRAMMER'S GUIDE TO RESERVED MS-DOS® FUNCTIONS AND DATA STRUCTURES

ANDREW SCHULMAN, RAYMOND J. MICHELS,
JIM KYLE, TIM PATERSON, DAVID MAXEY, RALF BROWN

Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sidney Singapore Tokyo Madrid San Juan Paris
Seoul Milan Mexico City Taipei

Library of Congress Cataloging-in-Publication Data

Undocumented DOS : a programmer's guide to reserved MS-DOS functions and data structures / by Andrew Schulman (general editor)... [et al.].
p. cm.
Includes bibliographical references (p. 679) and index.
ISBN 0-201-57064-5
1. MS-DOS (Computer operating system) I. Schulman, Andrew.
QA76.76.O63U53 1990
005.4'46—dc20 90-46992

Managing Editor: Amorette Pedersen
Copy Editor: Barbara Tilly
Set in 10.5-point Palatino by Benchmark Productions

3 4 5 6 7 8 9 10 - MW - 94939291
Third printing, June 1991

Table of Contents

Introduction

There is a story behind this book:

For months, a coworker of mine, having been misinformed that I was some sort of "DOS hotshot," had been pestering me to write a program to convince MS-DOS that it no longer had an L: drive. I never quite figured out what the program was for, but apparently customers were clamoring for a way to remove the Microsoft CD-ROM Extensions (MSCDEX) from memory, and this program had something to do with it. Anyway, I had tried several different approaches, including using MS-DOS's Cancel Device Redirection function (INT 21h Function 5Fh Subfunction 04h), all without success.

I then came across Ralf Brown's "Interrupt List," part of which now forms the appendix to this book. One of the DOS functions listed by Ralf, Get List Of Lists (INT 21h Function 52h), was marked "DOS 2+ internal." It was not listed in the official DOS documentation: IBM's DOS 3.3 *Technical Reference* jumps straight from Find Next (INT 21h Function 4Fh) to Get Verify Setting (Function 54h), with no mention of any functions between 4Fh and 54h. Even Ray Duncan's *Advanced MS-DOS Programming* simply lists Function 52h as "Reserved." Anyway, the undocumented DOS function described in Ralf's list turned out to be exactly what was needed to write a program to "cancel" the L: drive. Once I knew about INT 21h Function 52h, writing DRVOFF (as the utility was called) was trivial. Without this information, it was impossible.

This introduction is not the place to go into the details of DRVOFF. The source code for a similar program, and a full explanation, appear in chapter 4 of this book. For now, the point is simply that *here was a very real need that could be met only using a DOS function that doesn't appear in Microsoft's or IBM's documentation.*

Of course, I was already aware, like most DOS programmers, that there are undocumented DOS functions. In fact, I had accumulated a folder marked "Undocumented DOS" that contained various clippings from *PC Magazine, Dr. Dobb's Journal,* and *Programmer's Journal,* printouts of source files downloaded from bulletin boards, and printouts of discussions held in on-line conferences such as ibm.dos/secrets on BIX. I also found a number of books that contained discussions of undocumented DOS. But its randomness bothered me, and led to this book. The snippets I had collected were not only random, they were also sometimes contradictory, and they never seemed to take sufficient account of differences between DOS versions. It was clear that something more comprehensive was needed: a book that attempted to list *all* undocumented DOS functions and data structures in one place, detailing the DOS version differences (even for such odd versions of DOS as the compatibility box in OS/2), that frankly acknowledged the quirks of working with undocumented DOS, and that showed how to use undocumented DOS *safely* when writing programs.

Who better to write such a book than the software engineers who had already written elsewhere on the subject? Jim Kyle was an obvious choice, because he had prepared the material on undocumented DOS in the second edition of Que Corporation's popular *DOS Programmer's Reference.* Ray Michels was too, because he had written the chapter on "Undocumented DOS" in the Waite Group's *MS-DOS Papers.* Ralf Brown maintained the one truly definitive, absolutely reliable list of DOS calls, and it was clear that this "Interrupt List" had to be transformed into the book's appendix. Tim Paterson had not so much written *about* MS-DOS; he in fact wrote MS-DOS itself (Version 1.0) and was the perfect person not only to describe one important facet of undocumented DOS, but also to act as technical advisor for the entire book. It was also clear that the book needed to give readers a utility that would allow them to explore DOS without disassembling, and David Maxey, toolsmith extraordinaire, was the obvious choice to write INTRSPY.

What You Will Find in This Book

Most programmers who have worked with a DOS technical reference have probably wondered at some time or other about its curious "holes": function numbers that are marked "Reserved" or even entirely missing. This book contains a detailed version-by-version explanation of these "missing" DOS functions.

In addition, the book emphasizes the crucial undocumented DOS *data structures*, such as Memory Control Blocks (MCBs), the Current Directory Structure (CDS), Swappable Data Area (SDA), and the List of Lists. It also details the undocumented fields in structures that are otherwise documented, including the Program Segment Prefix (PSP), File Control Block (FCB), Drive Parameter Block (DPB), BIOS Parameter Block (BPB), and so on. For any of this to be usable in real programs, you should pay careful attention to our descriptions of how the layout of these structures differs from one DOS version to the next.

But more than simply listing the undocumented functions and data structures, this book provides *techniques*. Some of these already belong to the "folklore" or "oral history" of DOS programming. Others appear here for the first time. Here are a few of the techniques you will find in this book:

- Accessing the master environment
- Walking the DOS memory chain
- Loading device drivers from the DOS command line
- Creating logical drives with the network redirector
- Adding new internal commands with INT 2Fh Function AEh
- Writing TSRs with the DOS swappable data area

We have also tried, where appropriate (for example, accessing the master environment), to discuss *several different* techniques for performing the same task. This serves two purposes: first, to show the advantages and disadvantages of each technique, and second, to suggest that *safe* use of undocumented DOS might include performing the same operation in two different ways and then comparing the results to make sure they match.

The programs for this book were tested extensively in MS-DOS and PC-DOS versions 2, 3, 4, as well as a version that we can't talk about yet, but which may be available by the time you read this. The programs were also tested in such *simulated* DOS environments as the "compatibility box" in OS/2 1.1 and 2.0 and Digital Research's DR-DOS. Surprisingly, we found that our programs, which rely heavily on undocumented DOS, tended to work across a wider range of DOS and

pseudo-DOS versions than many programs that use only the documented DOS interface. Relying on undocumented DOS did not mean throwing caution to the winds. In fact, it meant we had to do a *better* job of DOS version checking than many programs that simply assume they are running under DOS 3.x or greater.

Undocumented DOS: The Disks

What will you find on the disks that accompany *Undocumented DOS*?

For one thing, it's not just the electronic form of the source files printed in the book (don't you just hate when publishers do that?). Certainly, the disks *do* contain all the code printed in the book. However, there's a lot of added value as well. For example:

- INTRLIST—Ralf Brown's famous "Interrupt List" in hypertext form, prepared by WindowBook, Inc. of Cambridge, MA. In addition to all the undocumented DOS functions and data structures that appear in print in Appendix A of this book, INTRLIST also includes all the documented calls in convenient on-line form, plus hard-to-find information on key DOS extensions, including NetBIOS, DPMI, DESQView API, Novell NetWare API, and so on.
- INTRSPY—David Maxey's script-driven debugger for monitoring PC software interrupts, described in chapter 8 of this book, plus many sample INTRSPY scripts.
- DEVLOD—Jim Kyle's program for loading device drivers from the DOS command line. Very handy, unless you actually *like* editing CONFIG.SYS and rebooting.
- ENVEDT—Jim Kyle's program for editing the master environment.
- MONITOR and WINMON—Complete assembly language source code for DOS and Windows debuggers, written by Tim Patterson.

Isn't This Material Secret?

> "I believe I undertook amongst other things not to disclose any trade secrets. Well, I am not going to."
> —Joseph Conrad, *Heart of Darkness* (1899)

None of the material in this book is particularly secret. Some of it has been available in one form or another in computer magazines or on electronic bulletin boards. What makes this book different is that we have brought all this scattered

material together in one place and have supplied tons of code examples showing how to actually *use* the material.

Probably all of the authors have at one time or another had a nondisclosure agreement with Microsoft, but none of the material in this book is based on anything Microsoft told us under nondisclosure. We did indeed undertake not to disclose any trade secrets, and we haven't.

Some aspects of undocumented DOS in fact constitute an open secret, well-known by anyone who cares to know, but still "reserved" by Microsoft and IBM. Sometimes this reaches heights of absurdity, such as when Microsoft's own "Dr. Bob" in *Microsoft Systems Journal* (September, 1987) discussed the well-known undocumented "InDOS" function (INT 21h Function 34h), yet still stated that the function is "undocumented, and unsupported by Microsoft." When Microsoft's own publications discuss undocumented DOS, there's certainly no reason for us not to discuss it, too.

On the other hand, this book includes much material unavailable elsewhere. The network redirector interface (INT 2Fh Function 11h) is apparently not even documented within Microsoft itself. *Undocumented DOS* also contains the first discussion we've seen of the important DOS swappable data area (SDA), of the installable command interface (INT 2Fh Function AEh), or of using the normal DOS termination function (INT 21h Function 4Ch) to deinstall a memory-resident program.

What Do We Mean By Undocumented DOS?

By undocumented DOS, we mean the body of functions and data structures that can reasonably be considered part of MS-DOS or PC-DOS but that are either not mentioned in the Microsoft or IBM documentation or that are marked "Reserved."

Deciding what is part of DOS, though, isn't always easy. Obviously, INT 21h Functions 50h through 53h are part of DOS, but what about the DOS network redirector? MS-Windows? PC LAN? Should we include undocumented interrupts used by the Microsoft C run-time library? Undocumented OS/2 calls like DosQProcStatus()? Undocumented Intel instructions like LOADALL? In short, where do you draw the line?

We decided to take a fairly narrow definition of undocumented DOS. We also decided to try to include only genuinely undocumented material, and not just information that is hard to come by. In any event, the INTRLIST database on the

accompanying disk has all sorts of material that the reader might otherwise wish we had included. For example, you might wish we had included a chapter on INT 34h through INT 3Eh, used in Microsoft (and Borland) language products for floating point emulation, or INT 3Fh, used by the Microsoft overlay manager. We decided these didn't belong in this book, but they're on the disk.

On the other hand, this book does cover some *documented* DOS functions, because they have undocumented subfunctions (for example, INT 21h Function 4Bh Subfunction 01h), a corresponding data structure that contains undocumented fields (for example, FCBs), undocumented side effects under certain circumstances (for example, INT 21h Function 13h), and outright bugs (for example, INT 21h Function 4Ah).

Pandora's Box and Information Hiding

This is a good place to express some reservations about *Undocumented DOS*. All of the authors have used undocumented DOS in real-world programs, but we've done so only when the documented DOS interface didn't supply what we needed. We would like to caution the reader not to use undocumented DOS simply because it is there. Sure, go ahead and try out all the functions to see if they work. Write tons of sample programs, or modify ours. But before using these in a program on which others rely, please think twice: are you *sure* there isn't a way to do it using documented DOS function calls?

Our goal in writing *Undocumented DOS* was actually to introduce some order into the world of DOS programming. We hope that, rather than rely on a random collection of clippings, you will now be able to turn to a single, reliable source of information on undocumented DOS. But we're also a little worried about opening Pandora's box. Is our little book going to spawn a generation of programs that make massive use of undocumented DOS? Will Microsoft suddenly be unable to make improvements to DOS, because too many programs will rely on silly undocumented features that therefore have to be preserved?

But that's *already* a problem. Too many important programs already use undocumented DOS. Microsoft is even forced to recreate undocumented DOS so that key programs will run in the "compatibility box" of OS/2. Our book can hardly make this situation any worse.

Still, this points to a problem. Programmers should not have to use undocumented functions to do their job. In 1972, David Parnas put forward his now-famous design principle of "information hiding." In a way, "information hiding"

dictates that software systems *must* have undocumented, hidden features, and that such undocumented features are a good thing, not a bad thing. When an interface is designed properly, programmers should have *no need* to use or even know about these hidden features: everything they need to do their job is supplied by the interface itself.

Thus, "information hiding" relies on a contract of sorts: the system promises to supply everything you need to write robust programs, and you in turn promise not to look below the surface. The internals of the system can then be improved or otherwise changed without affecting your program. Microsoft can come out with DOS 5, in other words, and your program written for DOS 2.x will still run. That's how "information hiding" is *supposed* to work. The problem is that MS-DOS *doesn't* give software developers everything they need, forcing them then to rely on machine-dependent or undocumented features.

Maybe it's okay that DOS doesn't supply everything developers need, though. In fact, I'm convinced this is one source of its success. Operating systems that attempt to provide all possible functionality have been far less successful than MS-DOS which, after all, barely merits the label "operating system."

No one can doubt MS-DOS's success. A *People* magazine profile of Bill Gates asserts that MS-DOS runs on 50 million machines worldwide. Although this figure sounds a little inflated (by comparison, there are probably only about ten times that many motor vehicles worldwide), the fact remains that the sheer size of the DOS marketplace is in itself a key aspect of MS-DOS. The size of this market means that software developers can afford to *force* DOS to do their bidding, and if this means using undocumented DOS, ignoring the principles of information hiding, and opening Pandora's box, then so be it.

Who Are You?

Readers will get the most mileage from this book if they are already familiar with DOS programming—that is, with how to make INT 21h calls. However, it is possible that many readers will be curious about undocumented DOS even when they are not completely comfortable with the *documented* DOS programmer's interface. Therefore, chapter 2 includes a brief review of the basics of calling DOS.

Readers will also get more out of this book if they know C or assembly language. Again, however, chapter 2 does include code samples in both Turbo Pascal and BASIC as well, so this can serve as a Rosetta Stone, allowing the reader to translate discussions that use C and assembly language into more familiar terms.

The bottom line is that readers have to be programmers familiar with the IBM PC and compatibles. The only chapters that could conceivably interest a nonprogrammer are chapter 1 (which discusses general issues regarding undocumented DOS, such as which commercial software uses it) and chapter 6 (which discusses the DOS command interpreter, COMMAND.COM).

Who Are We?

Having discussed who you are, and what background knowledge you need to benefit from this book, it's now time for that most enjoyable task, talking about ourselves:

Ralf Brown has delved into the innards of MS-DOS and IBM PC compatibles since early 1984 and is well-known in the on-line community for maintaining the "Interrupt List" and writing a number of programs, including a communications program called RBcomm and a DESQview API library called DV-GLUE. He is a Ph.D. candidate in the School of Computer Science at Carnegie Mellon University, specializing in natural language understanding. Ralf may be contacted at ralf@cs.cmu.edu (Internet), ucbvax!cs.cmu.edu!ralf or harvard!cs.cmu.edu!ralf (UUCP), or >INTERNET:ralf@cs.cmu.edu (CompuServe).

Jim Kyle has been a professional writer since 1948 and has published more than a dozen books and hundreds of magazine articles. His most recent books include Que's *DOS Programmer's Reference* and *Using Assembly Language* (both originally written by others; Kyle revised them for their second editions) and coauthorship of four sections in the authoritative *MS-DOS Encyclopedia* (Microsoft Press). His recent articles have appeared in *Computer Language* magazine. Kyle has been studying operating systems since 1970 or so, on mainframes and minicomputers as well as microcomputers, including GCOS (mainframe), TRAC and RSTS (mini), and CP/M and MS-DOS (micro). Kyle has been Primary Forum Administrator of *Computer Language*'s forum on CompuServe since 1985 and has been professionally involved in software and systems design since 1967. He is currently one-quarter of the Graphics Development staff at Norick Software, Inc. Jim may be contacted on CompuServe at 76703,762.

David Maxey, author of INTRSPY, manages a network software development team in Cambridge, MA. He has more than 12 years' experience in consultancy and systems development, ranging from small business applications to main-

frame text database projects for the European Commission. Maxey studied Electrical Engineering at Imperial College in London.

Raymond J. Michels has been working with the MS-DOS operating system since its introduction. He wrote the chapter on "Undocumented MS-DOS Functions" for The Waite Group's *MS-DOS Papers* (Howard W. Sams) and an article on "Undocumented DOS Internals" for *Programmer's Journal* (1989). Ray is an independent consultant specializing in MS-DOS application and system programs. He can be contacted on BIX as rmichels.

Tim Paterson is the original author of MS-DOS, versions 1.x, which he wrote in 1980-1982 while employed at Seattle Computer Products and Microsoft. In 1983, he founded his own company, Falcon Technology, which manufactured and sold hard disk products. Falcon was eventually sold, becoming part of Phoenix Technologies, the ROM BIOS maker. In 1988, Paterson left Microsoft (again), where he had been on the QuickBASIC 4.0/4.5 development team. He is now an independent consultant and has written several articles for *Dr. Dobb's Journal*, including the two-part series "Managing Multiple Data Segments Under Microsoft Windows" (with Steve Flenniken) and "Assembly Language Tricks of the Trade." He has a B.S. in computer science, magna cum laude, from the University of Washington.

Andrew Schulman is a software engineer and writer at Phar Lap Software (Cambridge, MA), makers of 386 | DOS-Extender. He is a contributing editor to *Dr. Dobb's Journal*, where he has written extensively about protected-mode DOS extenders and about OS/2. He is a coauthor of the book *Extending DOS* (Addison-Wesley, 1990) and has also written for *Byte* and *Microsoft Systems Journal*. He may be contacted at andrew@pharlap.com (Internet), uunet!pharlap!andrew (UUCP), or on CompuServe at 76320,302.

Acknowledgments

We would like to take a moment to do something more enjoyable even than talking about ourselves; to thank those who, knowingly or unknowingly, helped us put together *Undocumented DOS*.

First, a big round of applause for the on-line community, including participants in the ibm.dos/secrets (and secrets.2 and secrets.3) conference on BIX, and to the many contributors to the "Interrupt List" maintained by Ralf Brown. The names of the contributors to this list can be found in the INTRLIST database on disk, but we would like to single out Richard Marks (rmarks@KSP.Unisys.COM), Duncan Murdoch (dmurdoch@watdcsu.waterloo.edu), Robin Walker (rdhw@uk.ac.cam.phx), and Wes Cowley (wes@cup.portal.com), all of whom contributed major pieces of information on undocumented DOS.

Now, onto our technical reviewers. As editor of *Undocumented DOS*, I would like to thank Tim Paterson, who not only wrote a great chapter on the DOS debug interface, but who acted as technical advisor for the entire book. He provided key elaboration to several chapters in this book and cleaned up our assembly listings.

Dan Spear of Quarterdeck convinced us that the undocumented approach to LASTDRIVE in chapter 2 was actually *better* than the documented approach, and he told me all about a Novell NetWare quirk. Rob Adams of Phar Lap Software bugged me about MCB chains, Bob Moote (author of 386 | DOS-Extender and a member of the DPMI Committee) helped debug the DPMI sample code at the end of chapter 2, and Richard Smith (president of Phar Lap) suggested some of the techniques for *safe* use of undocumented DOS. Ben Williams of Rational Systems (makers of Instant-C and DOS/16M) also read parts of the book, as did Drew Grislagon of Datalight (makers of ROM-DOS, a DOS-equivalent operating system for embedded systems).

This book would not have happened without Claudette Moore. Open almost any good Microsoft Press book (including the *MS-DOS Encyclopedia* and *Advanced MS-DOS Programming*), and you will see Claudette's name. Now she is in charge of the Moore Literary Agency. Claudette rounded up authors, worked on the book's outline, sent out contracts, secured a publisher, sent out contract amendments, helped meet the deadline, sent out more contract amendments, and called in every other day "just to see how things are going." Thank you, Claudette, and congratulations on your wedding!

Thanks to all the folks at Addison-Wesley and Benchmark Productions. Chris Williams and Amy Pedersen in particular claim to know almost nothing about computer science, yet they are experts in the fields of *pipelining* and *parallel processing*. Pieces of this book were already emerging, typeset in attractive Palatino and OCRB, even before some chapters had entered the pipeline. It is not at all clear to me how they managed to turn this book around so quickly, even while producing several other books at the same time: apparently the Addison-Wesley algorithm for parallel processing is undocumented.

Finally, as editor of *Undocumented DOS*, I would like to thank my wife, the writer Amanda Claiborne, and my son, three-year-old Matthew Jacob Schulman, for providing me with the extra time needed to finish this book.

Andrew Schulman
Cambridge, MA
August, 1990

Chapter 1

Regarding the Use of Undocumented DOS

Andrew Schulman

The MS-DOS operating system for IBM PC and PS/2 computers and compatibles is the most widely used operating system in the world. One estimate puts the number of commercial and internally developed corporate applications for MS-DOS at more than 20,000. Estimates of the installed base of DOS systems range from 30 million to 50 million. This is a very wide range, and some of these estimates appear in marketing literature, so let's be conservative and call it 30 million. That's far more users than any other operating system.

On each of these 30 million machines, MS-DOS (or PC-DOS, as it is also called) provides not only its familiar user interface of the A> or C> prompt, but also a programmer's interface. Just as users make DOS requests by typing commands such as "DIR *.EXE" or "SUBST F: C:\SWAP," so programs make DOS requests—to open a disk file, to allocate memory, or even to terminate—by moving a function number into the Intel processor's AH register and issuing the assembly language instruction INT 21h. The MS-DOS programmer's interface consists of several software interrupts, most importantly INT 21h.

Just as MS-DOS itself is everywhere, technical documentation on how to program this ubiquitous piece of code turns up everywhere, too. Starting with the bible of DOS programming, Ray Duncan's superb *Advanced MS-DOS Program-*

ming (Redmond, WA: Microsoft Press, 1988), information about DOS programming is readily available. In fact, it is almost *too* available: a medium-sized bookstore might carry half a dozen different books on how to make INT 21h calls. Can there really be that many DOS programmers out there?

Most DOS programming books, after a few chapters on input/output, disks and files, memory allocation, and perhaps error handling or compatibility/performance tradeoffs, contain a lengthy appendix listing the INT 21h calls. These books start with INT 21h Function 0 (Terminate Process), proceed to Function 1 (Character Input With Echo), then to Function 2 (Character Output), and then, not surprisingly, to functions 3, 4, 5, and so on.

Clearly, MS-DOS is a well-ordered world, where all available functionality is carefully spelled out in numerous books that are readily available. MS-DOS is very small compared to many other computer operating systems, so it is possible to grasp DOS programming in its entirety. In contrast to the unfathomed depths of larger operating systems such as UNIX, MS-DOS is apparently a small, static world, in which everything there is to know already *is* known.

Well, not quite.

Open an official reference to the MS-DOS programmer's interface, for example the IBM DOS 3.30 *Technical Reference*, and you will find that the INT 21h function numbers jump straight from 4Fh (Find Next) to 54h (Get Verify Setting), with nothing at all said about the numbers in between. Even Duncan's *Advanced MS-DOS Programming* simply lists Functions 50h through 53h as "Reserved."

If you now turn to Appendix A of this book, you will find entries for the following functions:

INT 21h Function 50h—DOS 2+ —Set PSP Segment
INT 21h Function 51h—DOS 2+ —Get PSP Segment
INT 21h Function 52h—DOS 2+ —Get List Of Lists
INT 21h Function 53h—DOS 2+ —Translate BIOS Parameter Block

This is just one of many crucial "holes" in the programmer's interface to MS-DOS. Another hidden area of DOS is Function 5Dh, which consists of 12 subfunctions that handle an assortment of tasks, including DOS calls over a network (Server Function Call) and support for DOS reentrancy (Get Address of DOS Swappable Data Area). Although MS-DOS really is a small piece of code, it is nonetheless far from being a self-enclosed, static world. This small piece of code contains many uncharted areas.

Even some of the INT 21h functions that are documented have undocumented subfunctions (for example, Function 4Bh Subfunction 01h loads a program without executing it, and is crucial for writing a DOS debugger). Other functions have undocumented behavior or side effects (for example, documented Function 56h exhibits interesting behavior when invoked indirectly via undocumented Function 5Dh). Some functions have—dare we say it?—outright bugs (for example, look at the entry for INT 21h Function 4Ah in Appendix A).

Besides INT 21h, there are other DOS software interrupts, such as INT 2Fh, which contains entire undocumented subsystems such as the Network Redirector (INT 2Fh Function 11h) and the programmer's interface to APPEND.EXE (INT 2Fh Function B7h).

Actually, these "missing" functions are merely the most apparent portion of undocumented DOS. The real core of undocumented DOS is its data structures: undocumented fields in the Program Segment Prefix (PSP), the Drive Parameter Block (DPB), the DOS internal variable table (List of Lists), the Memory Control Block (MCB), the System File Table (SFT), and numerous other structures that are described in detail in this book.

Why Leave Functionality Undocumented?

> "Secrecy for plans is needed, not only to protect their formulation but also to develop them, perhaps to change them, at times to execute them, even to give them up."
> —Sisela Bok, *Secrets: On the Ethics of Concealment and Revelation* (1983)

At first glance, it seems absurd for Microsoft Corporation, the developer of MS-DOS, not to document all areas of the operating system. After all, what is the point of having functionality, if you don't tell people about it?

However, all software of any complexity must contain features that its developers choose not to bring out into the open. Once a software developer documents some feature of a product, it is almost obligated to support that feature in future releases. Microsoft has enough problems being required to maintain features of MS-DOS that *are* documented—the persistence of such CP/M-compatible anachronisms as File Control Blocks (FCBs) and the structure of the DOS Program Segment Prefix (PSP) are good examples—without also having to make sure that the *internal* structure of DOS is preserved, too.

Sometimes Microsoft's documenting a feature has downright unfortunate results. For example, in DOS 1.0 Microsoft documented the fact that, in addition to using INT 21h, applications could call operating system functions with a CALL 5 instruction. This DOS holdover from CP/M was used by several then important programs, including WordStar. MS-DOS supported CALL 5 by placing a far JUMP instruction at offset 5 in the PSP. Because this and other silly fields in the PSP were documented, every DOS program, even when running on the hottest new 80486 machine, gets loaded with a PSP that seems to harken back to the days of CP/M and 64KB memory. By making change more difficult, documenting features creates anachronisms.

From Microsoft's perspective, then, it makes perfect sense to reserve entire areas of DOS, and to tell developers that if they somehow find out about these areas and use them, their programs might or might not work in future releases. Microsoft has a standard policy statement about programs that use undocumented DOS functions and data structures:

> Title: Regarding the Use of Undocumented MS-DOS Features
> Document Number: Q34761 Publ Date: 5-SEP-1988
> Product Name: Microsoft Disk Operating System
> Product Version: 1.x 2.x 3.x 4.00
> Operating System: MS-DOS
> **Summary:**
> Microsoft does not give out any information about undocumented system features. If calls, flags, or interrupts are undocumented, it is because they are not supported; we can give NO guarantee that they will exist in future releases of DOS. If you find out about these features (through articles or by chance) and begin using them in your programs, there is a real potential that your application will not work in future DOS versions. We strongly advise against using undocumented features for these reasons and will give out no information about their use.
>
> Copyright Microsoft Corporation, 1989.

This is a reasonable statement, but there are other possible views on this subject. This chapter argues that PC programmers should know about undocumented DOS functions and data structures. The chapter explains why such undocumented features are necessary to fulfill MS-DOS's potential as an extensible operating system, tries to dispel some of the mystique surrounding undocu-

mented DOS, describes some of the important commercial software that uses undocumented DOS, and discusses some of the pros and cons of using undocumented features in application programs.

Why Is Undocumented DOS Important?

Why do we even care about undocumented DOS? What difference does it make whether the INT 21h function numbers are consecutive?

One reason, of course, is pure curiosity. Any time a table or a function is marked "Reserved," it raises questions: Why? Reserved for whom?

By itself, curiosity is not a good reason for exploring undocumented DOS. Most processors, for example, have "reserved" bits whose value you really shouldn't depend on or even care about. Later, this chapter explains why using undocumented DOS is completely different from relying on undocumented hardware features. For now, though, let's only consider why you should even *care* about undocumented DOS.

The real reason for discussing undocumented DOS is the importance of MS-DOS itself—remember those 30 million machines across the globe that run MS-DOS. At one point, Microsoft attempted to supplant MS-DOS with OS/2, an operating system with many wonderful features, but with a nearly insatiable appetite for memory and hardware. Since then, Microsoft has had to acknowledge that, warts and all, DOS is here to stay, probably to be *supplemented*, not replaced, by OS/2. MS-DOS continues to grow in importance. Therefore, even the tiniest piece of new information about DOS programming is potentially important to many programmers and, ultimately, to many users. This book presents many large chucks of new information about programming the world's most widely available operating system.

It still seems unlikely that there could be anything genuinely new to say about MS-DOS. After all, it is a piece of code that is in actuality quite small. The two components of the DOS kernel (IO.SYS and MSDOS.SYS), together with the replaceable COMMAND.COM shell, total at most 110KB of code. How can it require an entire industry—books, magazine articles, electronic bulletin boards, and user's groups—to explain less than 110KB of code? The electronic manuscripts for some of the chapters in this book were larger than that!

The key is DOS's *extensibility*. In fact, the small size of DOS seems to enhance its reach, not diminish it. MS-DOS provides few services. Some have even declined to refer to it as an operating system at all, referring to DOS instead as a

mere "program loader." But DOS's small size leaves room for extensions, and the services it does provide allow it to be extended in numerous directions, few of them anticipated by the system's original designer. That is a sure indication of a successful design, or at least an indication that such a large market exists for MS-DOS software that it is economically feasible for companies to invest the blood, sweat, and tears necessary to make this glorified program loader do their bidding. Either way, MS-DOS has been not only enormously successful but also enormously extensible.

What sort of extensions are we talking about? The best-known examples are memory-resident or terminate-and-stay-resident (TSR) programs, but other DOS extensions include:

- Windowing systems
- Multitaskers
- Networks
- Installable file systems (for example, CD-ROM)
- Debuggers
- Protected-mode DOS extenders

So we have a wildly successful operating system that can be extended in more or less any direction the marketplace seems to want. What more could we ask for? Why look for more, previously undocumented, functionality? Don't we have everything we need?

Permission, But Not Support

The problem is that many of the DOS functions and data structures that Microsoft has not documented are crucial to fulfill MS-DOS's potential as an extensible operating system. Notice that we have been saying that DOS allows or permits almost infinite extensibility: we never said that DOS actually *supports* such extensions. That is because support, as opposed to mere permission, tends to reside in the undocumented areas of the DOS programmer's interface.

No TSR Support

The field of memory-resident software is a good example of an area that permits extensions but that does not support them. MS-DOS allows programs to install interrupt handlers and to stay resident. The three documented INT 21h functions 25h (Set Interrupt Vector), 31h (Terminate and Stay Resident), and 35h (Get Inter-

rupt Vector) are sufficient to hook into, modify, or replace even INT 21h itself. This is an extremely powerful capability: nothing in DOS prevents you from taking over INT 21h.

But nothing particularly supports you in that endeavor, either, and that's the problem. There are documented functions that let a DOS program install itself as part of the operating system, but the functions that actually help the application behave *properly* once it is resident are undocumented. These include INT 21h Function 34h, 50h, and 51h, plus INT 28h.

That TSR support is confined to undocumented areas of MS-DOS is by now notorious. As far back as 1986, representatives from Microsoft sat down with representatives from other companies to work out an industry standard for TSRs, and one of the topics discussed was undocumented DOS. According to the *Microsoft Systems Journal*:

> "Currently TSRs depend on several undocumented MS-DOS features such as the IN_DOS flag . . . , the critical error flag, and some undocumented system calls. Microsoft's Adrian King has agreed to provide this information. Both Borland and Lotus say that this information is critical for TSRs to work consistently" (Nancy Andrews, "Moving Toward an Industry Standard for Developing TSRs," *Microsoft Systems Journal*, December 1986, pp. 10–11).

As Ray Michels explains in more detail in chapter 5 which covers TSRs and DOS multitasking, the DOS functions most critical to consistent TSR operation are as follows:

- INT 21h Function 34h (Return InDOS Pointer)
- INT 21h Function 50h (Set PSP Segment)
- INT 21h Function 51h (Get PSP Segment)
- INT 21h Functions 5D06h, 5D0Bh (Get DOS Swappable Data Area)
- INT 21h Function 5D0Ah (Set Extended Error Information)
- INT 28h (Keyboard Busy Loop)

To this day, Microsoft has still not added these to the official MS-DOS programmer's interface. In DOS 3.0 and higher, Function 51h is no longer strictly necessary, because an equivalent Function 62h (Get PSP Address) was added, but the other functions remain unsupported.

Microsoft has occasionally discussed undocumented DOS support for TSRs. In addition to the *Microsoft Systems Journal* article just quoted, the 1,500-page *MS-DOS Encyclopedia* (Redmond, WA: Microsoft Press, 1988) includes a fine chapter

on TSRs by Richard Wilton that describes most of the functions just mentioned. However, all but one of the functions were still omitted from the book's reference section, and the one function that was included (INT 21h Function 34h) bore the note "Microsoft cannot guarantee that the information in this entry will be valid for future versions of MS-DOS."

By now, information on undocumented DOS TSR support is fairly widely available, and it is well known that, to write *correct and stable* TSRs, you must use undocumented functions. Far from producing unreliable software, in the somewhat twisted land of DOS, undocumented functions are sometimes *necessary* to produce reliable software!

Network Redirector

Another area that permits extensions but that does not support them is the DOS file system. Anyone who has used a PC on a network knows how disk drives on another machine, perhaps not even a PC running DOS, can be made to appear like a local disk drive. You might type "DIR E:," for instance, to see the names of files on a Macintosh (truncated to fit DOS's pathetic 8.3 filename space). How does that work? How are all the INT 21h calls necessary to produce a directory listing sent over the network to another machine, and how can you write such software yourself?

That this is not necessarily just a network issue is shown by the Microsoft CD-ROM Extensions (MSCDEX), a fascinating piece of software that uses undocumented DOS file system features to make a CD-ROM appear like a normal DOS device. Obviously, there must be some features in DOS that allow you to write fiction, as it were: taking a CD-ROM with the High Sierra or ISO-9660 file system and making it look as though it were a standard DOS device with a File Allocation Table (FAT) file system.

Again, Microsoft has issued snippets of information. An article by then Microsoft spokesman Tony Rizzo ("MS-DOS CD ROM Extensions: A Standard PC Access Method," *Microsoft Systems Journal*, September 1987, pp. 54–62) reveals that MSCDEX designates the drive letters it assigns to CD-ROM device drivers, not as local drives, but as remote, *network* drives, even though the CD-ROM player is probably sitting on the disk next to the computer, not connected to it via a network (though the second scenario is also possible, as in the case of Lotus CD/Networker). According to Rizzo, MSCDEX uses a component of MS-DOS called the "network redirector." Microsoft has never documented the network

redirector, but chapter 4 of this book explains it in detail, showing that networks and installable file systems (IFSs) use the network redirector in part by writing an interrupt handler for INT 2Fh Function 11h.

In this case, it is at first less clear that undocumented DOS is absolutely necessary. After all, Novell has been producing reliable, high-performance networks for MS-DOS since long before Microsoft added the network redirector. Rather than hook INT 2Fh Function 11h, Novell hooks INT 21h itself. Although this avoids use of the undocumented network redirector, however, NetWare simply uses other undocumented features of DOS.

Support for Debugging

One last example: one thing you need in order to write a DOS debugger like DEBUG, Symdeb, CodeView, or Turbo Debugger is a function that loads a program without executing it. DOS provides this as Subfunction 01 to INT 21h Function 4Bh (EXEC), and it is used in all three generations of the Microsoft debugger. Unfortunately, the official MS-DOS technical references simply list Function 4Bh Subfunctions 00 and 03; Subfunction 01 is undocumented.

Fear of the Undocumented

We can see that DOS includes a lot of undocumented functionality. Microsoft doesn't document these features, because it wants the freedom to change or discard them in future versions of DOS. Armed with *Undocumented DOS*, you now know all about these functions and data structures. It's interesting to know that MS-DOS will return the address of its internal variable table if you invoke INT 21h Function 52h, or that EXEC Subfunction 1 loads a program without executing it. But can you really use this stuff in real programs?

Of course, Microsoft says no. So do other programmers as well. After all, in many areas of computing, the use of reserved, undocumented, or unspecified features is a one-way ticket to unstable, nonportable software. Use of undocumented features is not generally part of any approved software engineering curriculum. It is hard to believe that using undocumented features is often the only way to write stable and correct MS-DOS TSRs, network drivers, and debuggers.

There is a certain mystique surrounding undocumented DOS, and some programmers have found it easiest to take the dogmatic view that programmers should never, never use undocumented DOS functions in programs they plan to distribute to others.

For example, the author of a well-written, well-organized, and enjoyable introduction to TSR programming, writes:

> "None of the programs in this book use the INDOS call, and for good reason. INDOS is 'undocumented,' a term that has two meanings. The first is, of course, that you cannot look it up in the DOS manual. The second is that Microsoft, the vendors of DOS, reserve the right to change or delete this function from subsequent versions of DOS. In fact, the INDOS call as shown here is useful only under DOS version 2.x (where x is any of the minor version numbers). In DOS version 3.x the call still exists, but has changed quite a bit from the older versions. In DOS 4.0, this function does something quite different; thus, calls to the INDOS function will fail miserably.
> "For that reason, use of the INDOS function call or any undocumented DOS function is not recommended" (Thomas A. Wadlow, *Memory Resident Programming on the IBM PC*, Reading, MA: Addison-Wesley, 1987, p. 239).

In fact, the appendix to Wadlow's book contains entire pages with only a note at the top such as:

```
AH = 034H (52)                Unsupported
INT 021H (33)                 Universal function
```

with the rest of the page left blank!

A lot more can be said about INT 21h Function 34h than that. If you have all the information about changes made from one DOS version to the next, then calls to the INDOS function will *not* "fail miserably." Ray Michels' TSRs from Chapter 5 of this book use Function 34h and other undocumented DOS functions and data structures, but these programs work correctly in DOS 2.x, 3.x, 4.x and higher, in the DOS box of OS/2, and in Digital Research's DR DOS.

Thus, use of undocumented DOS does not necessarily prevent a program from running in the widest possible range of the DOS family. By following some of the techniques spelled out in this book, you can safely use INDOS and other undocumented DOS features in real programs. Such programs will have almost as good a chance of running properly in future versions of DOS than programs that restrict themselves to only documented DOS functions.

But you don't have to take our word for it. Many of the most successful commercial programs on the PC use undocumented DOS. We saw earlier that Lotus and Borland claimed that undocumented DOS was "critical for TSRs to work consistently." Of course they, like we, would prefer this functionality to be docu-

mented and supported by Microsoft. But in the meantime, far from adopting a hands-off policy regarding undocumented DOS, these companies make careful use of it.

An informal poll seems to indicate that developers of commercial PC software—software that must maintain a minimum of reliability and compatibility, sometimes on millions of different machines—are in general *less* fearful of undocumented DOS than programmers whose work needs to run only on one or two machines. A curious paradox.

One possible explanation is that programmers who work for large commercial software houses can better afford the possible higher cost of working with undocumented DOS. Perhaps software that uses these functions requires more testing and more maintenance than "normal" above-board DOS code. Another possibility is that it is mostly "system software" that requires undocumented DOS, and that most programs really don't require this stuff.

In any event, attitudes toward undocumented DOS resemble current opinions about the "goto" construct in programming languages. Many professional programmers recognize that goto, possibly disguised as "longjmp," is sometimes necessary. Undocumented DOS, like goto, should be avoided as long as possible, but not when its use becomes unavoidable. Software construction involves tradeoffs and compromises, not fixed dogmas. Software construction aspires to be engineering, not religion.

Reserved and Undocumented 80x86 Features

The fact remains, however, that in many other areas of computing, "reserved" features really shouldn't be tampered with. The fear of using reserved MS-DOS functions stems in part from a confusion with these other areas. Let's look more closely at one example: the practice of relying on reserved processor bits. We will see that there is a large difference between using undocumented DOS and using reserved or undocumented aspects of the Intel microprocessors.

At first, it certainly sounds as if undocumented DOS and what we might call "undocumented assembly language" could be considered the same thing. Intel's standard statement regarding undocumented assembly language sounds similar to Microsoft's statement, quoted earlier in this chapter, on undocumented DOS:

> "Depending upon values of reserved or undefined bits risks making software incompatible with future processors that define usages for those bits. Avoid any software dependence upon the state of reserved or undefined bits" (Intel, *386 DX Microprocessor Programmer's Reference Manual*, 1990, pp. 1–7).

Likewise, the following statement, from a brilliant survey of high-end microprocessors, sounds like it could just as well be talking about the use of undocumented operating system features: "When you see 'reserved' in a reference manual it really means that you should pay attention to it—it's very wrong to stomp on it." (Robert Dewar and Matthew Smosna, *Microprocessors: A Programmer's View*, New York: McGraw-Hill, 1990, p. 129). The example the authors give is quite instructive: In all the literature on the Intel 8088, INT 05 was marked as "reserved." When putting together the PC, however, IBM decided to use INT 05 as the ROM BIOS Print Screen function. Along came the 80286, and Intel (which had, after all, long before marked INT 05 as "reserved for use by Intel") picked INT 05 for the bounds exception interrupt.

The PC software industry is still cleaning up the resulting mess. INT 05 is actually just one of several cases where Intel says INT XX means one thing and IBM says it means something completely different. IBM's use of reserved Intel interrupt numbers was a hideous mistake.

If nothing else, this should caution us against thinking that simply because a major company does something seemingly "down and dirty," the practice in fact is necessarily safe. IBM totally blew it with INT 05. Are we about to do the same when we incorporate calls to INT 21h Function 52h in our programs?

No. IBM took a number that Intel reserved for future expansion and used it for its own purposes. But we're not proposing, as Microsoft has marked Function 52h as "reserved," that you go ahead and use it for your nifty new Dial Modem or Clear Screen function. Instead, we are saying that Function 52h *already has* a purpose, that what it does is in fact an "open secret," and that—assuming you exercise some precautions, as detailed in chapter 2 of this book—INT 21h Function 52h can be used in commercial software. That it *has* been used in commercial software does provide some added reassurance.

Undocumented assembly Language

Now, there is an aspect of "undocumented assembly language" that seems closer in spirit to undocumented DOS, and that is the use of undocumented Intel instructions, or the use of undocumented side effects of instructions. For example, the AAD and AAM instructions have "undocumented extensions," which can be used to multiply or divide by something other than ten.

Tim Paterson, author of this book's chapter on debugging (but perhaps better known as the author of MS-DOS 1.x itself), has written that relying on undocu-

mented features of the Intel 80x86 family "is a very dangerous practice . . . . There are too many different processors in the family—and too many different manufacturers—to consider using undocumented features. Let's all play by the rules." (*Dr. Dobb's Journal*, May 1990, p. 8).

So what's the difference between using undocumented aspects of the AAD instruction, on the one hand, which Tim deplores as a "very dangerous practice," and using undocumented DOS Function 4Bh Subfunction 01h, on the other hand, which he regards as essential for any DOS debugging, and to which he has devoted a chapter of this book?

Several key differences exist. First, there are several different manufacturers of 80x86 compatible processors, and there is no guarantee that all chips from NEC, AMD, and Harris will contain the same undocumented features as the Intel chips. In contrast, Microsoft is the only manufacturer of DOS that truly matters. (In any case, as we will see later, the manufacturers of DOS emulators have taken great pains to preserve its undocumented features.) And although Microsoft does make MS-DOS available to so-called original equipment manufacturers (OEMs), it is worth noting that the standard OEM license was apparently revised for MS-DOS 3.0 to require that OEM versions of MS-DOS not alter its internal data structures, which are required by SHARE and the network redirector.

Second, the range of the 80x86 family is far wider than that of DOS versions. This is perhaps unfortunate: it would be nice if somehow there were several varieties of DOS from which one could pick and choose ("okay, I'll use this one for my 8086 portable, and this one for the 80386"), but the fact is that there aren't. There are fewer differences between DOS 3.x and DOS 4.x than between, say, the 80286 and the 80386.

Third, Intel is quite simply less committed to preserving undocumented features in the 80x86 family than Microsoft is to preserving them in DOS. It's true that Microsoft won't openly support these functions, but Microsoft *is* committed to keeping DOS compatible with all the important PC applications. Furthermore, as you will see shortly, too many important programs rely on undocumented DOS for someone to seriously consider coming out with a version of DOS that *doesn't* provide at least the core undocumented DOS functions. In contrast, far fewer programs rely on undocumented assembly language.

Which brings us to the fourth point: fewer programs use undocumented assembly language than use undocumented DOS because it is *not necessary* to use undocumented assembly language tricks. The only possible use for using the

undocumented extension to AAD and AAM, for example, is not to perform an otherwise impossible operation, but to boost performance. In contrast, in *Undocumented DOS* we don't advocate the use of undocumented DOS for anything other than performing operations that would otherwise be impossible. Chapter 2 explains in detail that, given a combination of documented DOS functions that can do the same thing as an undocumented DOS function, you should use the documented interface. This is quite different from the motivation for using undocumented assembly tricks.

Finally, with the exception of multitaskers that use the Get PSP and Set PSP functions as part of context switching or debuggers that use the same two functions for switching between the debugger and the debuggee, most programs should not make many undocumented DOS calls. In particular, many undocumented DOS calls will be made *once*, at initialization time, when a program can afford to do rigorous checking of the DOS version number and perhaps a number of "sanity checks," such as those suggested in chapter 2 of this book. In assembly language, however, when undocumented CPU features are used to gain a few clock cycles, the undocumented feature is presumably being used in a block of code that is frequently called. Otherwise, why use it? If speed is the goal, however, branching to different blocks of code depending on the CPU (8088 versus 80286, etc.) is out of the question.

LOADALL

There *is* one undocumented Intel instruction that closely resembles our use of undocumented DOS and that *is* widely used in commercial software:

Go into a debugger, enter the bytes 0F 05 at CS:IP, and then unassemble CS:IP. In newer debuggers such as Microsoft CodeView 3.0 or Borland Turbo Debugger 2.0, you will see the name "LOADALL." Now, open one of the Intel programmer's reference manuals for the 80286 and higher. Find LOADALL? Didn't think you would. Even the "Opcode Map" just shows a blank for 0F 05, much like the holes in the DOS interface.

LOADALL is available only on the 80286 and can be used to access extended memory. A discussion of LOADALL itself is outside the scope of this book. What is relevant here is a discussion (from a book that is in many ways similar to this one) of whether or not to use LOADALL. This is analogous to discussions of whether or not to use undocumented DOS:

> "Assuming LOADALL is used cautiously, can it be used safely? That is, can we expect a program containing the LOADALL instruction to run correctly and reliably on a range of DOS versions, PC clone brands, and hardware configuration? The answer, at least on 80286-based PCs, seems to be a qualified *yes*. Microsoft uses LOADALL in the RAMDRIVE.SYS virtual disk driver supplied with Windows and the OEM versions of MS-DOS, and also uses it in the DOS compatibility environment of OS/2, so we can predict (given Microsoft's close relationship with Intel) that LOADALL isn't likely to vanish from future steppings of Intel's 80286 chips. For the same reason, the 80286 CPUs from second sources such as AMD and Harris will be obligated to support LOADALL indefinitely" (Ray Duncan, ed., *Extending DOS*, Reading, MA: Addison-Wesley, 1990, pp. 100–103).

This undocumented 80286 instruction is so important that all decent 80386 BIOSes contain emulation for LOADALL, which is missing from the 80386 chip itself. In fact, emulation of the LOADALL instruction is one way of judging whether an 80386 BIOS is truly "compatible." In the somewhat twisted PC world, an undocumented feature, far from being an obstacle to compatibility, can be essential to compatibility!

So once again we see how our industry's heavy-hitters make almost flagrant use of undocumented features. The next section takes a closer look.

Where Angels Fear to Tread: Programs That Use Undocumented DOS

What commercial software for the PC, including software written by Microsoft, uses undocumented DOS functions? We've already mentioned MSCDEX, DEBUG, Symdeb, and CodeView, but let's approach this in a more systematic way.

How do we find out what DOS functions, documented or undocumented, a program relies on? If we have access to the source code, we can just look at it. But disassembling programs like CodeView or MSCDEX violates your license agreement, and, furthermore, sounds as though it would be a pain.

Disassembling is also overkill, if you're just interested in what DOS calls a program makes. We said earlier that the architecture of MS-DOS lets you hook into system interrupts, including INT 21h itself. Why not write a utility that hooks INT 21h and other DOS interrupts and that tells you whenever a program makes a undocumented DOS call?

David Maxey's program INTRSPY was designed for this very purpose. It is an event-driven, script-driven DOS debugger that can also be used for many tasks having nothing to do with undocumented DOS. It is described in detail in chapter 8 of this book. You can write an INTRSPY script that logs information to

a file every time a program makes an undocumented DOS call. A very simple INTRSPY script that monitors undocumented DOS calls, but that doesn't use many INTRSPY features, looks like this:

```
; UNDOC.SCR (abridged version)
intercept 21h
    function 1fh on_exit output "211F: Get Default DPB: " DS ":" BX
    function 32h on_entry output "2132: Get DPB: " DL
    function 34h on_exit output "2134: InDOS flag: " ES ":" BX
    function 50h on_entry output "2150: Set PSP: " BX
    function 51h on_exit output "2151: Get PSP: " BX
    function 52h on_exit output "2152: Get List of Lists: " ES ":" BX
    function 53h on_exit output "2153: Translate BPB"
    function 55h on_entry output "2155: Create PSP: " DX
    function 5dh subfunction 06h
        on_exit output "215D06: Get DOSSWAP: " DS ":" SI
    function 60h on_entry
        output "2160: Canon File: " (DS:SI->byte,asciiz,64)
    function 4bh
        ; use this just to show which program made undoc DOS call
        subfunction 00h
            on_entry
                output (DS:DX->byte,asciiz,64)
        subfunction 01h
            on_entry
                output "214B01: EXEC debug: " (DS:DX->byte,asciiz,64)
    function 4ch on_entry output "-------------------------"
    function 31h on_entry output "--------- TSR -----------"
    function 25h
        on_entry
            if (al == 28h) output "SetVect INT 28h: KBD busy loop"
            ; not complete, because many programs unfortunately hook
            ; interrupts by poking the low-memory interrupt vector table
intercept 2eh
    on_entry output "2E: Execute command"
```

By loading INTRSPY into memory, feeding it UNDOC.SCR, running some programs, and then examining the report, you can see which programs use undocumented DOS. Let's try out the simple DOS utilities SUBST, JOIN, PRINT, CHKDSK, and APPEND:

```
intrspy
cmdspy compile undoc.scr
subst d: c:\swap
```

```
\dos33\join a: c:\floppy
print
chkdsk
append \undoc\intrspy
cmdspy report undoc.log
```

After you issue this series of DOS commands, the file UNDOC.LOG holds your report on undocumented DOS usage by some of the key utilities shipped with MS-DOS itself:

```
-------------------------
 C:\DOS33\SUBST.EXE
2152: Get List of Lists: 028E:0026
-------------------------
 C:\dos33\JOIN.EXE
2152: Get List of Lists: 028E:0026
2152: Get List of Lists: 028E:0026
2152: Get List of Lists: 028E:0026
-------------------------
 C:\DOS33\PRINT.COM
2151: Get PSP: 1376
2150: Set PSP: 1376
2152: Get List of Lists: 028E:0026
SetVect INT 28h: KBD busy loop
2134: InDOS flag: 028E:02CF
2150: Set PSP: 1376
2151: Get PSP: 1376
2150: Set PSP: 1376
2150: Set PSP: 1376
--------- TSR -----------
 C:\DOS33\CHKDSK.COM
2160: Canon File: C:\
2132: Get DPB: 03
2160: Canon File: C:\FLOPPY
-------------------------
 C:\DOS33\APPEND.EXE
--------- TSR -----------
 \undoc\maxey\CMDSPY.EXE
```

What this report shows you is that SUBST and JOIN both use INT 21h Function 52h. (For some reason, JOIN calls the function three times, which is probably unnecessary.) PRINT also uses Function 52h, but, more important, calls INT 21h Functions 50h, 51h, and 34h and hooks INT 28h—all necessary for PRINT's ability to multitask in the background. In addition to using Function 60h as a some-

what roundabout way of determining whether a DOS drive letter or directory corresponds to a physical device, CHKDSK also calls Function 32h to get the Drive Parameter Block (DPB). Finally, APPEND doesn't call *any* of the undocumented functions the program was monitoring (though it does use other undocumented functions, including the Installable Command facility provided by INT 2Fh Function AEh).

This script watches only *some* undocumented DOS functions because COMMAND.COM and DOS itself use so many that, if you watched them all, you could never tell which ones were being used by a program you were interested in and which ones were just part of the normal 3-degree background radiation of undocumented DOS calls.

Other Microsoft Software

Let's branch out now and look at some other Microsoft software: Windows 3.0, CodeView, and the Programmer's WorkBench from Microsoft C 6.0 (which, because of the performance of its real-mode DOS version, is also known as "Programmer's WasteBasket"). With INTRSPY still loaded in memory, and still processing the script UNDOC.SCR, run the following programs:

```
\win30\system\win /e
\c600\bin\cv \undoc\mem
\c600\bin\pwb \undoc\mem.c
cmdspy report undoc.log

------------------------
C:\WIN30\WIN.COM
C:\WIN30\system\win386.exe
2152: Get List of Lists: 028E:0026
2151: Get PSP: 40A3
2150: Set PSP: 40A3
2150: Set PSP: 40A3
2134: InDOS flag: 028E:02CF
2152: Get List of Lists: 028E:0026
2151: Get PSP: 40A3
215D06: Get DOSSWAP: 028E:02CE
 C:\win30\system\KRNL386.EXE
2134: InDOS flag: 028E:02CF
2151: Get PSP: 4215
2150: Set PSP: 40A3
2150: Set PSP: 4215
```

```
    .
 -------------------------
 c:\c600\bin\CV.EXE
2152: Get List of Lists: 028E:0026
2151: Get PSP: 40A3
2150: Set PSP: 0000
2150: Set PSP: FFFF
2150: Set PSP: 40A3
2150: Set PSP: 0E7B
2151: Get PSP: 0E7B
214B01: EXEC debug: C:\UNDOC\mem.EXE
2151: Get PSP: 441D
2150: Set PSP: 0E7B
    .
    .
-------------------------
 c:\c600\bin\PWB.COM
 c:\c600\bin\pwbed.EXE
    .
    .
 2152: Get List of Lists: 028E:0026
2151: Get PSP: 41D3
2151: Get PSP: 41D3
2151: Get PSP: 41D3
2150: Set PSP: 0000
2151: Get PSP: 0000
2150: Set PSP: FFFF
2151: Get PSP: FFFF
2150: Set PSP: 41D3
    .
    .
```

Examining the new entries in UNDOC.LOG, you see first of all that Windows 3.0 makes extensive use of undocumented DOS. In addition to retrieving the INDOS flag and the address of the List Of Lists, Windows 3.0 continuously calls the Get PSP and Set PSP functions in order to multitask between applications. Windows also uses INT 21h Function 5Dh Subfunction 06h to get the address of the "DOS swappable data area," whose structure is detailed in the appendix to this book and which is used in chapter 4 on the DOS file system, and in chapter 5 on TSRs. Microsoft CodeView also uses the Get PSP and Set PSP functions to switch between the debugger and the debugee, and it uses the undocumented EXEC Debug subfunction in order to load the debugee without immediately executing it. Finally, PWB also uses Functions 50, 51, and 52.

Other Software That Uses Undocumented DOS

By monitoring a series of popular PC commercial software from companies other than Microsoft, you can see how frequently undocumented DOS is used. With INTRSPY still processing UNDOC.SCR, run the following programs:

- SideKick (Borland)
- DESQView (Quarterdeck)
- Manifest (Quarterdeck)
- Norton Utilities
- DOS/16M (Rational Systems)
- 386 | DOS-Extender (Phar Lap)

```
-------------------------
 C:\SK.COM
2134: InDOS flag: 028E:02CF
-------------------------
 C:\DV\DV.EXE
2134: InDOS flag: 028E:02CF
2152: Get List of Lists: 028E:0026
2151: Get PSP: 168A
2150: Set PSP: 168A
2134: InDOS flag: 028E:02CF
2134: InDOS flag: 028E:02CF
2150: Set PSP: 168A
2150: Set PSP: 168A
2150: Set PSP: 168A
2155: Create PSP: 425E
2150: Set PSP: 425E
    .
    .
    .
-------------------------
 C:\QEMM\MFT.EXE
2152: Get List of Lists: 028E:0026
2134: InDOS flag: 028E:02CF
SetVect INT 28h: KBD busy loop
2134: InDOS flag: 028E:02CF
SetVect INT 28h: KBD busy loop
--------- TSR -----------
 C:\BIN\NU.EXE
2132: Get DPB: 03
2132: Get DPB: 03
-------------------------
 C:\BIN\NDD.EXE
2160: Canon File: A:CON
```

```
2160: Canon File: C:CON
2132: Get DPB: 03
2132: Get DPB: 03
------------------------
 C:\BIN\SD.EXE
2160: Canon File: A:CON
2160: Canon File: C:CON
2132: Get DPB: 03
2132: Get DPB: 03
------------------------
 C:\16M\LOADER.EXE
2152: Get List of Lists: 028E:0026
2152: Get List of Lists: 028E:0026
------------------------
 C:\PHARLAP\RUN386.EXE
2155: Create PSP: 8C40
2150: Set PSP: 1D7F
2150: Set PSP: 8C40
```

SideKick gets the address of the INDOS flag because the INDOS flag will tell it if it's safe to "pop up" (if the user activates SK's hot key and SK can't pop up, SK makes an odd chirping sound). The only surprise here is that there's no mention of the fact that SK hooks INT 28h: presumably it does so by poking the low-memory interrupt vector table instead of by calling documented INT 21h Function 25h. Sure, INT 28h is an undocumented interrupt, but that by itself is not a good reason to hook it in an underhanded, undocumented fashion as SK seems to do here.

DESQView, Quarterdeck's superb DOS multitasker, makes all the undocumented DOS calls that you by now expect of any DOS multitasking program. Quarterdeck's lovely diagnostic program, Manifest, explores many areas of undocumented DOS, so naturally it gets the List of Lists. If the user chooses to make Manifest memory resident, it gets the address of the INDOS flag and installs an INT 28h handler before going TSR.

Several key components of the Norton Utilities (NU), including the Norton Disk Doctor (NDD) and Speed Disk (SD) make the same undocumented DOS calls as CHKDSK.

Lotus 1-2-3 Release 3 incorporates a 16-bit protected-mode DOS extender, DOS/16M, from Rational Systems. In the absence of a VCPI control program such as Quarterdeck QEMM or Qualitas 386-to-the-Max, DOS/16M figures out whether VDISK or some other user of extended memory is loaded. The most reli-

able method of looking for VDISK is to use INT 21h Function 52h to find the head of the DOS device chain, and then to walk the device chain looking for VDISK.

Products such as IBM Interleaf Publisher, AutoCAD/386, Mathematica, and Paradox /386 all incorporate Phar Lap Software's 32-bit protected-mode DOS extender, 386 | DOS-Extender. The DOS extender makes two different undocumented DOS calls: a PSP is created for the protected-mode program, and then the Set PSP call is used to switch back and forth between the DOS extender and the protected-mode program itself. This is part of the mechanism that allows 32-bit protected-mode programs to call 16-bit real-mode MS-DOS.

So, there is a large collection of popular PC applications that use undocumented DOS. Are the vendors of all these programs going to get burned with the next version of DOS? It's instructive to read what Microsoft's Chief Architect for System Software says about this issue:

> "It may seem that if a popular application 'pokes' the operating system and otherwise engages in unsavory practices that the authors or users of the application will suffer because a future release, such as OS/2, may not run the application correctly. To the contrary, the market dynamics state that the application has now set a standard, and it's the operating system developers who suffer because they must support that standard. Usually, that 'standard' operating system interface is not even known; a great deal of experimentation is necessary to discover exactly which undocumented side effects, system internals, and timing relationships the application is dependent on" (Gordon Letwin, *Inside OS/2*, Redmond, WA: Microsoft Press, 1988, pp. 20–21).

In other words, when popular applications use undocumented DOS, it's ultimately Microsoft that is inconvenienced, not the application's developer. Smaller developers, meanwhile, can "ride the coattails" of the larger developer's use of undocumented DOS. If enough important applications use it, yesterday's undocumented hack becomes tomorrow's *de facto* "standard." The market has spoken. Amen.

Ain't Misbehavin'

As the previous section showed, many popular PC programs, mostly falling into the category of system software, use undocumented DOS. With the important exception of the multitasking programs PRINT, Windows, and DESQView, all of

which make very frequent use of the Get PSP and Set PSP functions as part of their context-swapping, these programs make *very few* undocumented DOS calls. This is somewhat like losing one's virginity, however: it takes only one undocumented DOS call to change the nature of a program.

Let's say that you start using one or two undocumented DOS calls in your program. What type of program do you now have? The chapter on "Compatibility and Portability" in Duncan's *Advanced MS-DOS Programming* categorizes MS-DOS applications by degrees of compatibility, and programs that use undocumented DOS are unequivocally exiled to the innermost circle of this DOS inferno:

> "'Ill-behaved' applications are those that rely on undocumented MS-DOS function calls or data structures, interception of MS-DOS or ROM BIOS interrupts, or direct access to mass storage devices (bypassing the MS-DOS file system). These programs tend to be extremely sensitive to their environment and typically must be 'adjusted' in order to work with each new MS-DOS version or PC model. Virtually all popular terminate-and-stay-resident (TSR) utilities, network programs, and disk repair/optimization packages are in this category" (second edition, 1988, p. 315).

The most important sentence here is the last one: if you write "ill-behaved" DOS applications, you will be in good company. Indeed, the purpose of *Undocumented DOS* is to show how you too can write such "ill-behaved" programs: programs like SideKick, the Norton Utilities, Windows, DESQView, and PRINT! All these programs *do* tend to be "extremely sensitive to their environment." Some of them *do* have to be "'adjusted' in order to work with each new MS-DOS version." Start using undocumented DOS, and that will be true of your software as well.

But that *already* is a fact of life in the MS-DOS world. In fact, using undocumented DOS has many of the same benefits and liabilities as the standard practice of bypassing DOS and writing directly to the hardware.

The need to use undocumented functions and data structures for many important tasks tells you much more about MS-DOS than it does about any sort of standard recommended engineering practice. Before we start knocking MS-DOS, though, let's not forget that, if for no other reason than that it has ridden on the coattails of the PC's wild success, DOS has succeeded in a way that no other, supposedly better, operating system can match. DOS, with all its warts, is an inescapable reality. Using undocumented DOS may not find a place in any software

engineering curriculum, but it is a good exercise in accommodating your principles to the real world.

Having said all this, let's see what we can salvage of good engineering practice as we make our descent into undocumented DOS. This book presents many techniques for using undocumented DOS in a relatively safe and reliable manner. Some of the techniques recommended in this book are:

- Rigorous checking of the MS-DOS version number
- Verifying the basic integrity of undocumented DOS internals by performing an undocumented DOS call and comparing its output with a known value
- Computing structure sizes dynamically as a double check for sizes computed from the DOS version number

Programs that use undocumented DOS are obligated to do a *better* job of DOS version checking, error checking and basic "sanity" checking than many other programs that otherwise play by the book. In fact, most of the programs in *Undocumented DOS* and its accompanying disk have been tested, and work properly, in MS-DOS versions 2.x, 3.x, 4.x and higher. Some of the programs have been ported to protected mode using DOS extenders. Many have been tested under different configurations, including Windows, DESQview, and QEMM and 386MAX with various DOS components loaded into high memory.

Simulated DOS

Many of the programs in this book have also been tested under environments such as the DOS compatibility boxes found in OS/2 1.x and 2.0, and Digital Research's DR DOS. These environments may or may not be important to you, but it is important to gauge the quality of their support for undocumented DOS, because any support they do provide *is completely intentional*. Unlike versions of MS-DOS itself, which may support one or another undocumented DOS feature simply out of inertia, these simulated DOS environments can only support an undocumented DOS function call or data structure if someone consciously *put* it there.

Let's look first at Digital Research's DR DOS 3.40, which provides an extremely close emulation of DOS 3.31 with SHARE.EXE loaded. So close, in fact, that many programs from this book run under DR DOS. We have already mentioned that Ray Michels' TSR from chapter 5 runs in this environment—that means DR DOS properly supports INT 21h Functions 34h, 50h, 51h, 5D06h, and

5D0Ah, and INT 28h. INT 21h Function 52h is of course supported, as is most of the DOS List of Lists, so MCB walkers and programs that walk the DOS device chain work just fine. Unfortunately, the DOS Current Directory Structure (CDS) is *not* supported, so many of the file system programs from chapter 4 of this book won't work in DR DOS 3.40. On the other hand, the oddball "installable command" functions discussed in Jim Kyle's chapter 6 on command interpreters *are* supported. Finally, INT 21h Function 60h is supported, but, in the version we examined, it had a bug (it always returned the name of the root directory, e.g., "C:\"). All in all, though, this product does an excellent job of emulation. Someone went to a lot of trouble to support the undocumented DOS interface, because that interface is essential to DOS compatibility.

The DOS compatibility boxes of OS/2 also provide an interesting perspective on undocumented DOS. The DOS box in the first release of OS/2 (so-called DOS 10.00) provided very little support for undocumented DOS. The *Microsoft Systems Journal* (May 1987) said that "since OS/2 does not recognize most of the undocumented MS-DOS services, programs that use them won't run in the compatibility mode." Lotus 1-2-3, Release 2.01 and dBase III Plus don't seem to make any undocumented DOS calls, so there probably didn't seem to be any good reason to support undocumented DOS. On the other hand, most popular TSRs *do* depend on undocumented DOS, so enough of undocumented DOS was supported so that SideKick would run in compatibility mode.

By the time of OS/2 1.1 and 1.2 (DOS 10.10 and 10.20), it was becoming clear that OS/2 was not going to replace DOS any time soon (and, in fact, might never replace DOS). Support for undocumented DOS was considerably beefed up. INT 21h Function 52h was supported and, for example, although most of the fields were set to FFFFh and you can't walk the DOS device chain (there is none!), you can walk the MCB chain. Most of Quarterdeck's Manifest can run properly in the OS/2 1.1 DOS box.

The forthcoming 32-bit OS/2 2.0 has greatly enhanced MS-DOS compatibility. In fact, it will allow you to run *multiple* DOS boxes, much as you can do today under DESQview or Windows 3.0. The support for undocumented DOS will be improved again. There seems to be a DOS device chain, and the LASTDRIVE field in the DOS List of Lists is supported, for example.

Again, the key point here is that this support for undocumented DOS isn't an accident. The market dynamics state that it has to be there: A DOS environment that can't support SideKick?! You must be kidding!

Categories of Undocumented DOS

It is helpful to try to take the large mass of undocumented DOS and break it into categories according to how reliable we think the different components are. A few undocumented DOS functions (in particular, INT 2Fh Function 12h) *are* unreliable in the sense that they were never meant to be called from outside the DOS kernel, so calling these functions from an application program is too tricky to be worthwhile. However, for all the features we've discussed in this chapter, reliability simply means how likely it is that the function or data structure will remain unchanged in future releases of MS-DOS.

In some cases, there appears to have been no good reason for the function to be undocumented in the first place, and the function has remained unchanged throughout its lifetime. Functions 50h (Set PSP) and 51h (Get PSP) are good examples of this category.

Function 52h (Get List of Lists) is an interesting case. The function itself has been remarkably stable and is relied upon by so many important applications that Microsoft would be foolish indeed to get rid of it. However, the List of Lists data structure itself has changed significantly from one DOS version to the next.

Get List Of Lists is probably the most important of all undocumented DOS functions, because with this single call you can access almost all of MS-DOS's internals. However, it is also easy to understand why this call is undocumented: the data structures it points to (either directly or indirectly) are the key data structures of MS-DOS itself. These *must* change when significant improvements are made to DOS. In some cases, new fields can be added to the end of a structure, so that none of its existing clients "break," and so that, over time, the data structure starts to resemble a "grab bag" (which is probably another reason it's undocumented; the interface is so messy it's embarrassing!) But in other cases, fields must be expanded or moved, and then any application that relies on a particular order or size of the undocumented data structures will break, and will need to be upgraded.

Therefore, it would be next to impossible for Microsoft to support use of this function in third-party software. It is probably difficult enough for Microsoft to keep all its *own* software that uses Function 52h happy from one DOS version change to the next. For example, hypothetically speaking, if the DOS 5 team needs to change a data structure pointed to by the List of Lists, and if Windows 3 relies on that undocumented data structure, what happens? If Windows 3 developers win the argument, this may prevent the DOS 5 developers from making an

important improvement. On the other hand, if the DOS 5 developers win the argument, a lot of Windows 3 update disks go out in the mail.

In any case, Functions 51h and 52h seem to belong to different categories of undocumented DOS. Is there some, more systematic, way to categorize undocumented DOS? One useful set of categories was drawn up by Ken W. Christopher, Jr., Barry A. Feigenbaum, and Shon O. Saliga, all IBM employees who were lead engineers for IBM's part in DOS 4. Their book, *Developing Applications Using DOS* (New York: John Wiley & Sons, 1990), is one source of information on undocumented DOS. Unfortunately, the book restricts its focus to PC-DOS 4.0. However, the author's DOS categorization (pp. 384–388) not only appears to be sensible, but, given that the authors work at IBM, may also reflect an insider's point of view:

"P Published interface. Will be supported in future DOS versions.

"O Obsolete function. Use the more modern function instead.

"X Excluded function. Do not use.

"U Unpublished function. Although not guaranteed by IBM or Microsoft to remain unchanged in the future, this function has been unchanged in DOS for several versions and is unlikely to change in future DOS versions.

"R Restricted unpublished function. This function should only be used when absolutely necessary to accomplish your program's function. This function is highly subject to change with each DOS version so your program should be both major and minor DOS version specific.

"D Implemented on Asian (DBCS) versions of DOS only."

Table 1-1 shows how the three IBMers apply these categories to undocumented INT 21h functions. The list uses their names for these functions, rather than the names used elsewhere in this book:

Table 1-1: Undocumented INT 21h functions from Christopher, Feigenbaum, and Saliga

Function	Description	Use
1Fh	Get Default DPB	UR
32h	Get DPB	UR
34h	GET INDOS Flag Address	U
37h	Get/Set Switch Character	U
4B01h	Load Program	U
50h	Set Active Process Data Block	U
51h	Get Active Process Data Block	U

Function	Description	Use
52h	Get DOS Internal Values	R
53h	Set DPB	UR
55h	Duplicate Process Data Block	UO
58h	Get/Set Allocation Method	U
5D00h	Server DOS Call	U
5D01h	Commit All Files	U
5D02h	Close File by Name	U
5D03h	Close All Files for a Particular Computer	U
5D04h	Close All Files for a Particular Process	U
5D05h	Get Open File List Entry	U
5D06h	Get DOS Data Area Address	RO
5D07/08h	Get/Set Print Stream State	U
5D09h	Truncate Print Stream	U
5D0Ah	Set DOS Extended Error Information	U
5D0Bh	Get DOS Data Areas	R
5E01h	Set Machine Name	U
5E04/05h	Set/Get Printer Mode	U
5F00/01h	Get/Set Redirection Mode	U
5F05h	Get Redirection List Entry Extended	U
60h	Translate Filespec	U
6520h	Capitalize Character	U
6521h	Capitalize String	U
6522h	Capitalize ASCIIZ String	U
6523h	Capitalize Yes/No Check	U
69h	Get/Set Media ID	R

The Case of the Missing One-Quarter

Here, we have only listed the functions in category U and R. What percentage of all INT 21h functions do these comprise? The three IBM authors list a total of 170 functions. Of these, 38 carry a U and/or an R. We might therefore conclude, only half jokingly, that MS-DOS is 22 percent undocumented. Oddly enough, this corresponds closely with our own findings: at one point, the complete "Interrupt List" found on the disk that accompanies this book was 747KB, and at the same time the electronic manuscript for Appendix A, which lists *only* undocumented DOS, was 180KB.

This means that we decided that about 1/4 of the material in the "Interrupt List" constitutes undocumented DOS. It's now time to take a closer look at this missing one-quarter of the PC programer's interface.

Chapter 2

Programming for Documented and Undocumented DOS: A Comparison

Andrew Schulman

This chapter looks at how to incorporate the information in the rest of the book into working code in C, 80x86 assembly language, Turbo Pascal, and BASIC. It also discusses the important issue of when *not* to use undocumented features, while showing that certain PC programming tasks absolutely require them.

We will also illustrate that exploiting undocumented features of MS-DOS usually requires only a few lines of code. On the other hand, programs that use undocumented DOS features must be more aware of the MS-DOS version number than code that uses only documented DOS. In particular, whereas undocumented MS-DOS *function calls* have remained remarkably stable from one version of DOS to another, the equally important DOS *data structures* vary significantly with each new release of the operating system, and programs that use undocumented DOS must take strict account of this.

At the end of the chapter, we will examine the issue of using undocumented DOS from protected-mode DOS extenders and from the DOS Protected-Mode Interface (DPMI), as found in Windows 3.0 386 enhanced mode.

Using Documented DOS Functions

Before examining how to use undocumented DOS in programs, let's review how to use *documented* DOS function calls. This detour into documented DOS (we might even say *over-documented* DOS, because so much has been written about it!) will pay off when we write programs using undocumented DOS.

If you know all about calling DOS from your chosen programming language, skip to the section on "Using Undocumented DOS."

If you're still here, let's pretend we work in the installation software group of a commercial software company. For some reason, we have been asked to produce a small utility that, when run from a DOS batch file, will return the number of "logical drives" on the system, corresponding to the LASTDRIVE statement in a user's CONFIG.SYS. Perhaps the company is installing software in a Novell NetWare environment, where LASTDRIVE determines the starting letter for network drives.

The utility is to be called LASTDRV.EXE, and the idea is that when it exits back to DOS, it should return a number corresponding to LASTDRIVE. For example, if LASTDRIVE=C, then LASTDRV.EXE should return the number 3. This is different from other DOS utilities that return 0 to indicate success and 1 (or more) to indicate an error. This number can be interrogated using the IF ERRORLEVEL facility in MS-DOS's somewhat demented batch language.

The LASTDRV utility should also display a string such as "LASTDRIVE=E," but in such a way that the output can be discarded by redirecting the program's output to the NUL "bit bucket" device.

For example, to make sure that there are at least six logical drives (LASTDRIVE is F: or higher), someone in the batch files team of the installation software group (large software companies really are organized that way!) would take our wonderful utility and incorporate it into the following batch file:

```
echo off
rem need6.bat
lastdrv > nul
if errorlevel 6 goto end
echo Requires at least six drives
:end
```

How do we write LASTDRV.EXE? Trying to find the user's CONFIG.SYS file and then locate the LASTDRIVE statement is a very bad idea. Aside from the fact

that LASTDRIVE didn't make its appearance until DOS 3.0 and that its use is optional (E: is the default LASTDRIVE), we would have no guarantee that, once we locate a CONFIG.SYS, it's actually the one with which the system was booted. It also appears to be impossible to locate the boot drive reliably in MS-DOS prior to version 4.0 (DOS 4 and higher do provide such a function, however: INT 21h Function 3305h).

If we are writing in a high-level programming language like C or Pascal, it's unlikely that the compiler's subroutine library comes with a function that returns the number of drives. True, Microsoft C has the function _bios_equiplist(), for example, and Borland Turbo C and Turbo C++ have the function biosequip(), which can be used to find the number of floppy drives. But what about fixed disks?

More important, we were asked to retrieve the number of DOS *logical* drives, so interrogating the PC's BOM BIOS does not meet the functional specification for this utility. "Logical" drives also include RAM disks, network drives, CD-ROM drives, tape back-up units, and the like. "Logical," in other words, means both physical drives and *fictional* drives. As shown in this book's chapter on the DOS file system, much of DOS's extensibility comes from the ability to have drive letters assigned to things that really aren't drives at all!

Thus, "logical drive" is an MS-DOS construct, having nothing to do with PC hardware or the ROM BIOS. To learn how a program finds out the value of LASTDRIVE, then, the first thing to do is browse through a reference book on the DOS programmer's interface, looking for an INT 21h function that returns the number of logical drives.

Flipping through any DOS programmer's reference, we find that INT 21h Function 0Eh, which is used to select the current disk drive in the system, somewhat illogically (the two have little to do with each other) also returns the total number of drives:

```
Int 21H Function 0EH
Select Disk
Selects the drive specified in DL (if valid)
as the default drive.
Call with:
    AH = 0EH
    DL = drive code (0=A, 1=B, etc.)
Returns:
    AL = number of logical drives in system
```

In single-drive IBM PCs in DOS 1.x and 2.x (the latter will be present at more customer sites than you would think), the value 2 is returned in AL, because DOS supports two logical drives (A: and B:) hanging off the same single physical floppy drive. Further, in DOS 3.x and higher, the value returned in AL is either 5 or the drive code corresponding to the LASTDRIVE entry (if any) in CONFIG.SYS, whichever is greater.

This return value is what we want. Actually, it's *almost* what we want. In one important special case—DOS machines using Novell NetWare—the value returned in AL by Function 0Eh is *not* equal to LASTDRIVE. Given the number of PC machines running Novell NetWare, this is indeed an important exception, and we will return to it later in this chapter.

How do we get back the return value without also selecting a new current drive? The answer is obviously to specify the drive that is *already* current as the "new" one. Where do we find the current drive? Once again we flip through our DOS programmer's reference (DOS programming has a lot in common with using a mail-order or gardening catalog!) until we stumble upon Function 19h (Get Current Disk):

```
Int 21H Function 19H
Get Current Disk
Returns the drive code of the current, or
default, disk drive.
Call with:
    AH = 19H
Returns:
    AL = drive code (0=A, 1=B, etc.)
```

It's really quite simple to take all this information and turn it into a program. In the remainder of this section, we will produce versions of LASTDRV.EXE in assembly language, C, Turbo Pascal, and QuickBASIC. Throughout, we will be using only thoroughly *documented* portions of the DOS programmer's interface, in preparation for our descent into the world of undocumented DOS.

DOS Calls From assembly Language

The following small assembly language program shows how the reference material on DOS Functions 0Eh and 19h translates into a working version of LASTDRV.EXE. This code also uses DOS Function 09h to display output on the screen; the output can also be redirected to a file or to the NUL "bit bucket."

Finally, DOS Function 4Ch is called to exit to DOS, passing the numeric value of LASTDRIVE as a return code:

```
; LASTDRV.ASM -- uses only documented DOS

_STACK  segment para stack 'STACK'
_STACK  ends

_DATA   segment word public 'DATA'
msg     db      'LASTDRIVE='
dletter db      (?)
        db      0dh, 0ah, '$'
_DATA   ends

_TEXT   segment word public 'CODE'

        assume cs:_TEXT, ds:_DATA, ss:_STACK

main    proc    near
        mov     ax, _DATA
        mov     ds, ax          ; set DS to data segment
        mov     ah, 19h         ; Get Current Disk function
        int     21h             ; call MS-DOS

        mov     dl, al          ; AL now holds current drive
        mov     ah, 0Eh         ; Select Disk function
        int     21h             ; call MS-DOS
        mov     bl, al          ; LASTDRIVE in AL; save in BL
        add     al, ('A' - 1)   ; convert to drive letter
        mov     dletter, al     ; insert into string

        mov     dx, offset msg  ; string in DS:DX
        mov     ah, 9           ; Display String function
        int     21h             ; call MS-DOS

        mov     ah, 4Ch         ; Return to DOS
        mov     al, bl          ; LASTDRIVE is exit code
        int     21h             ; call MS-DOS
main    endp

_TEXT   ends

        end main
```

LASTDRV can be assembled with any number of assemblers and then linked with any MS-DOS compatible linker:

Microsoft Macro Assembler (MASM):

```
masm lastdrv.asm;
link lastdrv.obj;
```

Borland Turbo Assembler (TASM):

```
tasm lastdrv
tlink lastdrv
```

Phar Lap 386 | ASM/LinkLoc:

```
386asm -8086 lastdrv
linkloc -8086 lastdrv
```

DOS Calls From C

There is a problem making DOS calls using the C programming language. It's not that it is difficult to access MS-DOS services from C: the problem is there are *too many* different ways to do so. Never satisfied with one technique where a dozen techniques will do, C compiler manufacturers for the PC, such as Microsoft, Borland, JPI, Watcom, and MetaWare (it is amazing that the PC marketplace apparently can support so many good C compilers), offer a wide variety of techniques for calling MS-DOS and ROM BIOS services. Having many different ways to perform the same operation is never a good idea.

The problem isn't really with the compilers, however. Ultimately, we have to ask why MS-DOS itself doesn't come with a set of standard include files, the way OS/2 does. On the other hand, this lack of standard programming facilities in MS-DOS has done nothing to stop MS-DOS's spectacular success, and may even have aided it slightly, because it gives programmers one more thing to manipulate. In any case, we need to discuss a few of the techniques that can be used to make MS-DOS calls from C, including the int86() and intdos() functions, in-line assembly language, and register pseudo-variables.

int86() Until recently, the most popular way of calling system services from C on the PC was to use the int86() family of functions, which invoke Intel 80x86 software interrupts:

```
/* LASTDRV.C -- uses only documented DOS */

#include <stdio.h>
#include <dos.h>
```

```
main(void)
{
    union REGS r;
    unsigned lastdrv;

    r.h.ah = 0x19;                  /* Get Current Disk */
    int86(0x21, &r, &r);            /* call MS-DOS */
    r.h.dl = r.h.al;                /* r.h.al now holds current drive */
    r.h.ah = 0x0E;                  /* Select Disk */
    int86(0x21, &r, &r);            /* call MS-DOS */
    lastdrv = r.h.al;               /* r.h.al now holds number of drives */
    fputs("LASTDRIVE=", stdout);    /* output string */
    putchar('A' - 1 + lastdrv);     /* output drive letter */
    putchar('\n');                  /* output newline */
    return lastdrv;                 /* return drive number to MS-DOS */
}
```

This can be compiled with any Microsoft-compatible C compiler for the IBM PC, using either the full-screen or the command-line version of the compiler. For example:

Microsoft C 6.0:

```
cl -qc lastdrv.c
```

Borland Turbo C++:

```
tcc lastdrv
```

The C source code is almost half the length of the corresponding assembly language code we examined earlier. On the other hand, the size of the executable file has grown from less than 600 bytes in assembly language to almost 5,000 bytes in C.

In-line Assembler A better way to write PC system-level software in C is to use an in-line assembler: that is, put Intel assembly-language code directly in your C code. True, an in-line assembler is inherently nonportable, but so are calls to int86(). You can't expect MS-DOS or ROM BIOS calls to work on non-Intel architectures anyway, so this is in fact a perfect place to use in-line assembly language.

Microsoft C 6.0, Microsoft Quick C 2.5, Borland Turbo C, and Borland Turbo C++ all include an in-line assembler. There is a slight difference between the Microsoft and Borland dialects. Microsoft offers an _asm block, whereas Borland requires that the asm keyword precede each line. Microsoft put a scaled-down assembler right into its C compiler, whereas Borland passes the in-line assembler

through to a separate assembler such as TASM or MASM, allowing you to include assembly-language directives (such as DB), assembly-language macros, or 386 instructions directly in your C code. Either way, the essentials are the same. Note how the preprocessor directives ensure the compiler can support an in-line assembler:

```
/* LASTDRV2.C -- uses only documented DOS */

#include <stdlib.h>
#include <stdio.h>

main()
{
    unsigned lastdrv;

#ifdef __TURBOC__
    asm mov ah, 19h          /* C-style comments only */
    asm int 21h
    asm mov dl, al
    asm mov ah, 0x0e         /* C-style hex */
    asm int 21h              /* assembly-style hex */
    asm xor ah, ah
    asm mov lastdrv, ax      /* refer to C variables */
#elif (defined(_MSC_VER) && (_MSC_VER >= 600)) || defined(_QC)
    _asm {
        mov ah, 19h          ; can include assembly-style comments
        int 21h              /* and C-style as well */
        mov dl, al           // and this style as well
        mov ah, 0x0E         ; can include C-style hex numbers
        int 21h              ; or assembly-style hex numbers
        xor ah, ah
        mov lastdrv, ax      ; can refer to C variables in _asm
        }
#else
#error Requires inline assembler
#endif

    fputs("LASTDRIVE=", stdout);
    putchar('A' - 1 + lastdrv);
    putchar('\n');
    return lastdrv;
}
```

The comments inside the _asm block show the odd mixtures of C and assembly language that can be produced.

You do have to be careful when using in-line assembly language. In particular, you must know your compiler's rules about preserving registers. For Microsoft C and Turbo C, the rules are simple: you are free to change AX, BX, CX, DX, and ES. Inside a function, you can change BP. You can change any flag except the direction flag. You must always preserve the DI, SI, DS, SS, and SP registers, however.

Register Pseudo-Variables Borland and JPI both provide yet another way of writing low-level code: register pseudo-variables. Not to be confused with C register variables, register pseudo-variables map onto the CPU registers but look like C variables:

```
/* LASTDRV3.C -- uses only documented DOS */

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

main()
{
    unsigned lastdrv;
    _AH = 0x19;
    geninterrupt(0x21);
    _DL = _AL;
    _AH = 0x0E;
    geninterrupt(0x21);
    lastdrv = _AL;
    fputs("LASTDRIVE=", stdout);
    putchar('A' - 1 + lastdrv);
    putchar('\n');
    return lastdrv;
}
```

Note that geninterrupt(0x21) is *not* a function call but a compiler directive to emit an INT 21h directly into the compiled code. Also, although the register pseudo-variables such as _AL are extremely handy, the code generated by the compiler uses the same CPU registers, so you can't rely on values staying in the registers for very long.

DOS Library Functions Actually, this exact same operation could have been performed without int86() or _asm blocks. Most C compilers for the PC provide a set of functions that map directly onto the most popular DOS functions. Microsoft C provides functions with names such as _dos_getdrive() and _dos_setdrive(), for instance, and Turbo C provides getdisk() and setdisk(). In Turbo C, the DOS lastdrive() function is thus equivalent to setdisk(getdisk()). (Again with the important exception of Novell NetWare, which we will be discussing later.)

DOS Calls From Turbo Pascal

What about other calling MS-DOS functions from other high-level languages? In some ways, it is much simpler to make these calls from other languages, such as Turbo Pascal, because you don't have to worry about which method to use. As noted earlier, having the wide variety of techniques available in C ultimately isn't so terrific, because programmers (and writers) end up spending too much time deciding which technique to use.

Calling DOS functions from Turbo Pascal requires the Dos unit, which includes the Registers variant record (similar to union REGS in C) and the MsDos() function:

```
{ LASTDRV.PAS -- uses only documented DOS }

program LastDrv;
uses dos;
var
    r : Registers;
    lastdrive : Word;

begin
    with r do begin
        ah := $19;          { Get Current Disk }
        MsDos(r);
        dl := al;
        ah := $0E;          { Select Disk }
        MsDos(r);
        lastdrive := al;
    end;
    Writeln('LASTDRIVE=', Chr(Ord('A') - 1 + lastdrive));
    Halt(lastdrive);
end.
```

Note that Pascal's `with` construct allows us to refer to fields of the Registers record as, for example, ah rather than r.ah.

Using the command-line version of Turbo Pascal, LASTDRV.PAS can be turned into LASTDRV.EXE by typing:

```
tpc lastdrv.pas
```

True to Turbo Pascal's reputation for producing extremely tight code, the resulting Turbo Pascal executable file is only 2KB. The smallest C version was about 4KB.

DOS Calls from BASIC

Finally, what about BASIC? The following version of LASTDRV displays the LASTDRIVE letter and returns the numeric value of LASTDRIVE to the DOS ERRORLEVEL:

```
REM LASTDRVIVE -- uses only documented DOS
REM $INCLUDE: 'QB.BI'

SUB DOSEXIT(errorlevel)
    CLOSE
    DIM Regs AS RegType
    Regs.ax = &H4C00 + errorlevel    ' Terminate Process
    CALL INTERRUPT(&H21, Regs, Regs)
    PRINT "this is never executed"
END SUB

DIM Regs AS RegType
Regs.ax = &H1900                     ' Get Current Disk
CALL INTERRUPT(&H21, Regs, Regs)
Regs.dx = Regs.ax
Regs.ax = &H0E00                     ' Select Disk
CALL INTERRUPT(&H21, Regs, Regs)
lastdrv = Regs.ax AND &HFF
PRINT "LASTDRIVE="; CHR$(ASC("A") - 1 + lastdrv)
CALL DOSEXIT(lastdrv)
END
```

To turn this source code into an executable file, you can use either Microsoft Quick BASIC or the Microsoft BASIC 6.0 compiler:

BASIC 6.0 compiler:

```
bc /o lastdrv.bas;
link lastdrv,,,qb.lib;
```

Quick BASIC (must produce a stand-alone executable file!):

```
qb lastdrv.bas /L qb.qlb
```

Using the BC /O switch or producing a stand-alone executable file from within QuickBASIC is mandatory. Surprising as it seems, Microsoft BASIC has no provision for returning exit codes to DOS. In order to return the value of lastdrv as the DOS ERRORLEVEL, LASTDRV.BAS uses the subroutine DOSEXIT(), which directly calls MS-DOS Function 4Ch (Terminate Process with Return Code) *and never returns*, thereby bypassing BASIC's normal exit routine. This will not work from an executable file that uses the BASIC run-time module (for example, BRUN60EP). In fact, directly calling INT 21h Function 4Ch from an executable that uses the BASIC run-time module can easily hang the machine.

There's another problem. Because we never return after calling INT 21h Function 4Ch, we do an end-run around BASIC's exit routine, and BASIC never gets to clean up after itself. The result is that the cursor is lost when we return to the DOS prompt. Thus, although this code shows how to make low-level system calls from Microsoft BASIC, it really isn't a useful piece of software. BASIC has many features going for it as a programming language, but returning exit levels to the operating system apparently is not one of them. (Microsoft's latest incarnation of BASIC, though—the Professional Development System (PDS) 7.0—finally does allow BASIC programs to set the DOS exit code.)

Using Undocumented DOS

Quarterdeck's expanded memory manager, QEMM, comes with a program called LASTDRIV.COM, one of whose uses is to report the value of LASTDRIVE. Interestingly enough, this program does not use documented Function 0Eh. Instead, it uses undocumented Function 52h. We won't see why until later. For now, though, the point is that if Quarterdeck can do it, so can you. Just as you eventually have to learn about direct screen writes or programming the 8259 interrupt controller to be a successful PC programmer, so you need to learn about the proper use of undocumented DOS.

What better place to start than with a program whose operation is well known to us: LASTDRV. In the next section, we will again show how to write the

LASTDRV utility in assembly language, C, Turbo Pascal, and QuickBASIC, this time using undocumented DOS. In particular, we will highlight the larger role that the DOS version number plays when using undocumented DOS.

We just went through the process of using a standard DOS programmer's reference like an office-supply catalog or a handbook of mathematical functions, trying to find a tool that would help us write the LASTDRV utility. We never found a single function called Last Drive, but we did find two functions (Get Current Disk and Select Disk) that could be used together to achieve the same effect.

In other words, we saw lastdrive() is similar to setdisk(getdisk()). But there's something illogical in this: why should DOS return the total number of drives when you set the current drive? MS-DOS presumably keeps the value of LASTDRIVE somewhere internally. Is there some way to find it?

If you leaf through Ralf Brown's appendix to this book you will find DOS's internal location for LASTDRIVE in the middle of a DOS data structure called the List of Lists. The following shows the format of the List of Lists:

Offset	Size	Description
DOS 2.x		
10h	BYTE	number of logical drives in system
DOS 3.0		
1Bh	BYTE	value of LASTDRIVE command in CONFIG.SYS (default 5)
DOS 3.1-3.3		
21h	BYTE	value of LASTDRIVE command in CONFIG.SYS (default 5)
DOS 4.x		
21h	BYTE	value of LASTDRIVE command in CONFIG.SYS (default 5)

The List of Lists, or DOS internal variable table, is probably the most important undocumented DOS data structure, and INT 21h Function 52h, which returns in the ES:BX register pair a pointer to the List of Lists, is probably the most important undocumented DOS function.

Note how the offset of the LASTDRIVE field within the DOS internal variable table changed from DOS 2.0 to DOS 3.0 to DOS 3.1. This is the sort of undocumented DOS behavior our programs will have to deal with. What the offset will be in future versions is anyone's guess, and that, of course, is the whole problem with using undocumented DOS features.

In future versions of DOS, the LASTDRIVE field might even disappear, breaking whatever programs depend upon its presence. The only comfort is that, should the DOS List of Lists be changed radically, not only will our own programs start to fail but practically all important Microsoft software will break, too! In fact, the reliance of key pieces of Microsoft software such as Windows 3.0 on the internal structure of DOS might make this internal structure less likely to change. However, that perhaps is too much to hope for in a large company, where the Windows 3.0 group might not even talk with the DOS 5 group.

In the midst of the changes to the position of LASTDRIVE within the List of Lists—and, if you look at the appendix entry for INT 21h Function 52h, massive changes throughout the List of Lists as a whole—one thing *has* remained constant: INT 21h Function 52h itself, which from DOS 2.0 onward has been as stable as any documented DOS function and which is supported even in simulated DOS environments such as the compatibility box of OS/2 1.10:

```
INT 21 - DOS 2+ internal - GET LIST OF LISTS
AH = 52h
Return: ES:BX -> DOS list of lists
```

No Magic Numbers

Because the List of Lists is so central to DOS programming, many books on the subject end up using INT 21h Function 52h somewhere in their sample source code. However, because of their authors' possibly guilty feelings about using undocumented DOS in the first place, these books simply leave the code unexplained. For example, from the Turbo Pascal source code in a useful book on LAN programming, the following appears without any explanation:

```
regs.ah := $52;
intr($21, regs);
ofs := regs.bx + $22;
seg := regs.es;
while memw[seg:ofs] <> $ffff do
```

The author needed to use INT 21h Function 52h because it is the only way to accomplish certain key tasks in DOS programming. To use this function and then not explain what it does, though, seems far worse than any explicit use of undocumented DOS. To use undocumented DOS and not explain it gives your code a mysterious quality. As used above, 52h and 22h are certainly "magic numbers." Let's see if we can't completely demystify INT 21h Function 52h.

You can try out this function, without even writing a program, by using the DOS DEBUG utility. First assemble the DOS call and execute it:

```
C:\UNDOC2>debug
-a
775A:0100 mov ah, 52
775A:0102 int 21
775A:0104 nop
775A:0105
-g 104

AX=5200  BX=0026  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=775A  ES=028E  SS=775A  CS=775A  IP=0104   NV UP EI PL NZ NA PO NC
775A:0104 90            NOP
```

The register dump shows that in this sample DEBUG session, ES:BX points to 028E:0026. As we will see later, from DOS 3.1 on, the DOS internal variable table actually starts at offset -12 (decimal) from the address returned in ES:BX. In this case, therefore, the table starts at offset 0026h - 0Ch (12 decimal), or 001Ah:

```
-d es:001a
028E:0010                                   03 00 01 00 00 00              ......
028E:0020  FF 0A 00 00 F3 09 F0 75-8E 02 98 00 8E 02 A4 01   .......u........
028E:0030  70 00 6E 01 70 00 00 02-00 00 49 0C 00 00 EE 0C   p.n.p.....I.....
028E:0040  00 00 6D 0A 00 00 03 05-12 00 F4 09 04 80 99 15   ..m.............
028E:0050  9F 15 4E 55 4C 20 20 20-20 20 00 90 43 17 8E 02   ..NUL     ..C...
028E:0060  47 17 8E 02 47 17 8E 02-43 17 8E 02 43 17 8E 02   G...G...C...C...
028E:0070  43 17 8E 02 43 17 8E 02-43 17 8E 02 47 17 8E 02   C...C...C...G...
028E:0080  43 17 8E 02 43 17 8E 02-43 17 8E 02 47 17 8E 02   C...C...C...G...
028E:0090  43 17 8E 02 43 17 8E 02-00 00                     C...C.....
```

Aside from the header for the NUL device driver, it is difficult to find our way around here. If we compare the DEBUG dump with the format of the List of Lists as shown in the appendix entry for INT 21h Function 52h, however, it all makes sense. We can even see that the Turbo Pascal code quoted earlier was adding 22h to the value returned from Function 52h so that it could get a pointer to

the NUL device, which is at the head of DOS's device chain. This is one of the most popular uses of INT 21h Function 52h, but clearly the DOS List of Lists holds many other goodies as well:

Offset	Size	Description	
-12	WORD	sharing retry count	0003
-10	WORD	sharing retry delay	0001
-8	DWORD	pointer to current disk buffer	0AFF:0000
-4	WORD	unread CON input	0000
-2	WORD	first Memory Control Block	09F3
00h	DWORD	first Disk Parameter Block	028E:75F0
04h	DWORD	list of DOS file tables	028E:0098
08h	DWORD	pointer to CLOCK$ device driver	0070:01A4
0Ch	DWORD	pointer to CON device driver	0070:016E
---DOS 3.1-3.3---			
10h	WORD	max bytes/block	0200
12h	DWORD	first disk buffer	0C49:0000
16h	DWORD	Current Directory Structures	0CEE:0000
1Ah	DWORD	pointer to FCB table	0A6D:0000
1Eh	WORD	number of protected FCBs	0000
20h	BYTE	number of block devices	03
21h	BYTE	LASTDRIVE	05
22h	18 BYTEs	actual NUL device driver header	[... NUL ...]
34h	BYTE	number of JOIN'ed drives	00

Because the value of the LASTDRIVE field in the List of Lists is 5, LASTDRIVE=E, which is the default value when CONFIG.SYS does not include a LASTDRIVE statement.

Having seen a little bit of what the DOS List of Lists looks like, we can now retrace our steps in building the LASTDRV utility, this time using INT 21h Function 52h and the LASTDRIVE field within the DOS internal variable table.

You may be thinking that this is a futile exercise, because we already know how to get the value of LASTDRIVE using a completely safe and documented function that doesn't change with each new version of DOS. However, we will see later on that using the undocumented internal value of LASTDRIVE can actually be *more* reliable than using the documented Function 0E return value (after

all, Quarterdeck must have *some* reason for using Function 52h instead of Function 0E!).

Undocumented DOS Calls From assembly Language

The following small assembly language program shows how the reference material on DOS Function 52h and the DOS List of Lists translates into a working version of LASTDRV.EXE. All use of undocumented DOS is confined within the subroutine _lstdrv:

```
; LASTDRV2.ASM -- uses undocumented DOS

        assume cs:_TEXT, ds:_DATA, ss:_STACK

_STACK  segment para stack 'STACK'
_STACK  ends

_DATA   segment word public 'DATA'
msg     db      'LASTDRIVE='
dletter db      (?)
        db      0dh, 0ah, '$'
_DATA   ends

_TEXT   segment word public 'CODE'

        public  _lstdrv

_lstdrv proc    far
        push    si
        push    bx
        push    cx

        mov     si, 1Bh         ; assume DOS 3.0

        mov     ax, 3000h       ; Get MS-DOS version number
        int     21h             ; major=AL, minor=AH
        cmp     al, 2
        jl      fail            ; Requires DOS 2+

        jne     dos3up           ; DOS 3+
        mov     si, 10h         ; DOS 2.x
        jmp     short get
dos3up: cmp     al, 3
        jne     ofs21
        and     ah, ah          ; DOS 3.0
```

```
        jz      get
ofs21:  mov     si, 21h         ; DOS 3.1+, DOS 4.x

get:    mov     ah, 52h         ; Get List of Lists
        xor     bx, bx          ; Zero out ES:BX so we can check
        mov     es, bx          ;   for NULL after INT 21h
        int     21h             ; list=ES:BX
        mov     cx, es
        or      cx, bx          ; Is ES:BX NULL?
        jz      fail            ; Function 52h not supported

        mov     al, byte ptr es:[bx+si]
        xor     ah, ah          ; return LASTDRIVE in AX
        jmp     short leave

fail:   xor     ax, ax          ; return 0 in AX
leave:  pop     cx
        pop     bx
        pop     si
        ret
_lstdrv endp

main    proc    near
        mov     ax, _DATA
        mov     ds, ax

        call    _lstdrv
        and     ax, ax          ; test for failure
        jz      done

        mov     bl, al          ; save LASTDRIVE in BL
        add     al, ('A' - 1)   ; convert LASTDRIVE to drive letter
        mov     dletter, al     ; insert into string

        mov     ah, 9           ; Display String
        mov     dx, offset msg
        int     21h

done:   mov     ah, 4Ch         ; Return to DOS
        mov     al, bl          ; exit code
        int     21h
main    endp

_TEXT   ends

        END     main
```

The main subroutine contains boring documented DOS code for displaying output and exiting to DOS. All the really interesting code is in the slightly convoluted _lstdrv subroutine, paraphrased in the following pseudocode:

```
offset := 1Bh;
ver := DosVersion();
if (ver.major < 2)
    return failure;
else if (ver.major == 2)
    offset := 10h;
else if (ver.major != 3 and ver.minor != 0)
    offset := 21h;
ListOfLists := GetListOfLists();
if (ListOfLists == NULL)
    return failure;
else
    return ListOfLists[offset];
```

The goal of the various DOS version number tests is to put the correct location of LASTDRIVE into the SI register, so that it can be added to the base address of the List of Lists that we get back from DOS undocumented Function 52h. The SI register is preloaded with the offset of LASTDRIVE for DOS 3.0, in an attempt to somewhat reduce the large number of JMPs.

Note how, in *all* DOS versions greater than 3.0, we will store 21h into the offset. When testing the DOS version number, it is generally useful to test for numbers *greater than or equal to* the highest known version (for example, version >= 4). Testing simply for equality (for example, version == 4) means that your application won't work in a future version such as DOS 5. It is amazing how many programs do DOS version checking incorrectly, thereby unknowingly cutting themselves off from future DOS versions.

By treating all DOS versions higher than 3.0 as one unit, obviously we are assuming that, for example, DOS 5 will store LASTDRIVE in the same place as DOS 3.3. When dealing with undocumented DOS, you can either make this assumption or you can take the more conservative approach of halting the program under unknown versions of MS-DOS. This "versionitis" is really the only problem with using undocumented DOS. If your application uses some of the less stable undocumented functions or data structures, perhaps you should use == rather than >= to test DOS version numbers. On the other hand, there *are* several double-checks your program could perform so that it is not simply left floundering

in the shifting sands of DOS internals; you will see several such double checks later in this chapter and in the next chapter.

When testing the DOS version numbers returned from DOS documented Function 30h, note that the *major* version number is counterintuitively returned in AL (the *low* portion of AX), and the minor version number is returned in AH (the *high* portion of AX). When testing DOS version numbers, it is also important to remember that a version number such as 3.1 is actually 3.10. In the case of DOS 3.10, the minor version number in AH is not 01h, nor 10h, but 10 decimal (0Ah).

In any case, once SI holds an offset appropriate to the version of DOS the program is running under, the rest is easy:

```
mov ah, 52h
int 21h
mov al, byte ptr es:[bx+si]
xor ah, ah
```

Actually, in LASTDRV2.ASM the code is slightly more complicated than this because we have taken the precaution of ensuring that undocumented INT 21h Function 52h is really supporting by checking that the pointer in ES:BX is not NULL. The ES:BX register pair is loaded with NULL prior to invoking INT 21h so that, in a really screwy simulated DOS environment that doesn't support this function, ES:BX will at least hold a reasonable value we can test for.

Note that we *don't* check whether the carry flag (CF) is set, however. Unless the documentation specifically says that a function sets or clears CF, the state of CF is undefined. The entry for INT 21h Function 52h in the appendix to this book says nothing about CF. Thus, far from being an extra-careful precaution, checking CF in fact would be a perfect example of relying on *undefined behavior*. As noted in chapter 1, using undocumented DOS is completely different from relying on undefined behavior.

Although this version of LASTDRV looks completely different from the version that used only documented DOS calls, the result is similar: the value of LASTDRIVE is both displayed and returned. The difference is that now we're getting our information straight from the horse's mouth, by examining the DOS internal variable table.

Undocumented DOS Calls From C

This book has spent so much time on the LASTDRV utility, and on various ways of performing DOS calls from C, that you would think there would be nothing new to say about making undocumented DOS calls from C. In fact, the following version (LASTDRV4.C) introduces a number of important topics, including the use of far pointers in C, the MK_FP() macro, testing the DOS version number in C, and the use of int86x() rather than int86():

```
/* LASTDRV4.C -- uses undocumented DOS */

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

#ifndef MK_FP
#define MK_FP(seg,ofs) \
    ((void far *)(((unsigned long)(seg) << 16) | (ofs)))
#endif

main()
{
    union REGS r;
    struct SREGS s;
    char far *doslist;
    unsigned lastdrv_ofs;
    unsigned lastdrv;

    /* get offset for LASTDRIVE within DOS List of Lists */
    if (_osmajor < 2)
        return 0;
    else if (_osmajor == 2)
        lastdrv_ofs = 0x10;
    else if (_osmajor == 3 && _osminor == 0)
        lastdrv_ofs = 0x1b;
    else
        lastdrv_ofs = 0x21;

    /* Get DOS Lists of Lists */
    r.h.ah = 0x52;
    segread(&s);
    s.es = r.x.bx = 0;
    int86x(0x21, &r, &r, &s);
    /* make sure Function 52h is supported */
    if (! s.es && ! r.x.bx)
```

```
        return 0;
    doslist = MK_FP(s.es, r.x.bx);

    /* Get LASTDRIVE number */
    lastdrv = doslist[lastdrv_ofs];

    /* OS/2 DOS compatibility box sets LASTDRIVE to FFh */
    if (lastdrv == 0xFF)
        return 0;

    /* Print LASTDRIVE letter */
    fputs("LASTDRIVE=", stdout);
    putchar('A' - 1 + lastdrv);
    putchar('\n');

    /* return LASTDRIVE number to DOS */
    return lastdrv;
}
```

If you contrast LASTDRV4.C with the earlier versions that used only documented DOS calls, you will notice a number of significant differences:

Rather than call INT 21h Function 30h to get the DOS version number, as we did from assembly langauge, we now use the global variables _osmajor and _osminor, provided by most C compilers for the PC. In Microsoft C, Watcom C 386, and MetaWare High C 386, these variables are declared in STDLIB.H; in Turbo C and JPI TopSpeed C, they are declared in DOS.H. It is important to remember that in DOS 3.3, for example, _osminor is 30 (decimal), not 3, and not 0x30, either.

Because DOS Function 52h returns the address of the List of Lists in ES:BX, and because int86() doesn't handle segment registers such as ES, we need to use int86x() and struct SREGS. We don't need to pass any segment registers *into* Function 52h, so it seems as though it doesn't much matter what values struct SREGS holds before calling int86x(). Nonetheless, it is a good habit to call the segread() function to load the struct SREGS, as we do here, because if you ever try to move your code to a protected-mode DOS extender, it will be crucial that the segment registers are never loaded with garbage values, even if these registers are seemingly not used.

Because the List of Lists is part of DOS, not located inside our program, it must be addressed with a four-byte (far) pointer. The C variable doslist is intended to hold this address, and is declared as a char far *, rather than as a char *.

This allows us to peek and poke DOS's internal variable table even from a C program that otherwise uses only two-byte (near) pointers.

After DOS Function 52h has returned the address of the List of Lists in ES:BX, int86x() returns it to us in s.es and r.x.bx. How do we move these into char far *doslist? LASTDRV4.C uses the macro MK_FP(), which (as its name implies) makes a far pointer from a segment and an offset. This handy macro is provided in the DOS.H include file with Turbo C and TopSpeed C but, unfortunately, not with Microsoft C. In LASTDRV4.C, we use the C preprocessor to define a MK_FP() macro if one is not already present. While the definition of MK_FP() makes it appear as if a shift left (SHL) is being performed, but in fact any good C compiler for the PC will turn this code:

```
void far *fp = ((void far *)(((unsigned long)(seg) << 16) | (ofs)))
```

into this:

```
mov ax, _seg
mov dx, _ofs
mov word ptr _fp, dx
mov word ptr _fp+2, ax
```

(You can examine your C compiler's output by compiling, for example, with the -Fa or -Fc switch in Microsoft C, or the -S switch in Turbo C.)

Rather than use the MK_FP() macro, in Microsoft C we could also use the following construct:

```
FP_SEG(doslist) = s.es;
FP_OFF(doslist) = r.x.bx;
```

FP_SEG() and FP_OFF() are two other important macros for PC systems programming in C. Whereas MK_FP() constructs a far pointer from a segment and an offset, FP_SEG() and FP_OFF() perform the opposite operation: FP_SEG() extracts the segment of a far pointer, and FP_OFF() extracts the offset. Microsoft's versions of FP_SEG() and FP_OFF() are a little strange in that they are C lvalues and can therefore be assigned to.

This C version of LASTDRV also does a bit more work than the assembly language version. Before printing out the LASTDRIVE letter, LASTDRV4.C checks to see if lastdrv is 0FFh. This is the value that the OS/2 1.10 DOS compatibility

box (also known as the "penalty box") uses for the LASTDRIVE field in the DOS List of Lists. A program running in this compatibility box thinks it is running under DOS 10.10, so you might think we should simply fail if (_osmajor >= 10). However, the support for undocumented DOS has improved in each version of the DOS box, so there is no reason to cut ourselves off unnecessarily from this simulated DOS environment. For instance, the DOS boxes in OS/2 2.0 (which masquerade as DOS version 20.0!) *do* provide proper support for LASTDRIVE, and for most other fields in the List of Lists as well. It is worth noting that, although the DOS version number is in the double digits, in fact the OS/2 compatibility box closely resembles DOS 3.10 with SHARE.EXE loaded.

What, No Structures? To most C programmers, the big question in LASTDRV4.C is "Where are the structures?!" You need only look at the entry for DOS Function 52h and the List of Lists in the appendix to see that all these offsets seem to cry out to be represented with a C structure. In fact, you might ask why this book doesn't present an UNDOC.H include file!

The reason we do not have an UNDOC.H include file for you is that programs that use undocumented DOS functions should use only a few of them. An UNDOC.H file could be an invitation to overuse undocumented DOS calls. We don't want to promote undocumented DOS as yet another "application programmer's interface" (API) consisting of several hundred "new" functions and data structures!

There is in fact an additional, more serious, problem with using data structures in undocumented DOS programming. This will become clear as we discuss the next program, LASTDRV5.C, which uses a C structure to represent much of the DOS List of Lists:

```
/* LASTDRV5.C */

#include <stdlib.h>
#include <dos.h>

#pragma pack(1)

#define LISTOFLISTS_DECR        12

typedef struct {
    unsigned shareretrycount;
    unsigned shareretrydelay;
    void far *currdiskbuff;
```

```
    void near *unreadcon;
    unsigned mcb;
    void far *dpb;
    void far *filetable;
    void far *clock;
    void far *con;
    union {
        struct {
            unsigned char numdrive;
            unsigned maxbytes;
            void far *first_diskbuff;
            unsigned char nul[18];
            } dos2;
        struct {
            unsigned char numblkdev;
            unsigned maxbytes;
            void far *first_diskbuff;
            void far *currdir;
            unsigned char lastdrive;
            void far *stringarea;
            unsigned size_stringarea;
            void far *fcbtab;
            unsigned fcb_y;
            unsigned char nul[18];
            } dos30;
        struct {
            unsigned maxbytes;
            void far *diskbuff;
            void far *currdir;
            void far *fcb;
            unsigned numprotfcb;
            unsigned char numblkdev;
            unsigned char lastdrive;
            unsigned char nul[18];
            unsigned numjoin;
            } dos31;    /* and higher */
        } vers;
    } ListOfLists;

main()
{
    union REGS r;
    struct SREGS s;
    ListOfLists far *doslist;
    unsigned lastdrive;

    /* No List Of Lists in DOS 1.x */
```

```
    if (_osmajor < 2)
        return 0;

    /* Get DOS List of Lists */
    r.h.ah = 0x52;
    segread(&s);
    s.es = r.x.bx = 0;
    intdosx(&r, &r, &s);
    if (! s.es && ! r.x.bx)
        return 0;
    doslist = MK_FP(s.es, r.x.bx - LISTOFLISTS_DECR);

    /* Get LASTDRIVE value, depending on DOS version */
    if (_osmajor == 3 && _osminor == 0)
        lastdrive = doslist->vers.dos30.lastdrive;
    else if (_osmajor == 2)
        lastdrive = doslist->vers.dos2.numdrive;
    else
        lastdrive = doslist->vers.dos31.lastdrive;

    /* print LASTDRIVE letter, return LASTDRIVE number */
    printf("LASTDRIVE=%c\n", 'A' - 1 + lastdrive);
    return lastdrive;
}
```

From looking over struct ListOfLists, you should understand why the DOS internal variable table is called the List of Lists: most of the fields are just pointers to other data structures, including the list of DOS Memory Control Blocks (MCBs), the list of Drive Parameter Blocks (DPBs), the DOS device chain, and the File Control Block (FCB) table. In fact, in a complete struct ListOfLists, these other fields, rather than using void far *, would each use, for example, FCB far * or DPB far *.

Within struct ListOfLists, a C union is used to manage the differences between DOS versions. Unions help represent the changes that each version of DOS brought to the List of Lists. Each component of a C union is allocated storage starting at the beginning of the union, and the size of a union is the amount of storage necessary to represent its largest component. In other words, as in a variant record in Pascal, the components are overlaid. In the union vers within struct ListOfLists, the same block of memory can be viewed as a struct dos2, a struct dos30, or a struct dos31.

The line that reads #pragma pack(1) is essential. By default, C compilers for the PC align structures on word (two-byte) boundaries. For our C structure to

correspond exactly with the layout of the DOS internal variable table, we need to pack the structure on byte boundaries. Otherwise, an unsigned char followed by an unsigned short would occupy four bytes, not three, and our structure would not reflect DOS's internal variable table.

Note that we create a far *pointer* to a struct ListOfLists, *not* a struct ListOf-Lists. The memory for the structure already exists inside DOS.

As noted earlier, rather than using ES:BX as a pointer to the List of Lists, we use ES:BX-12. The appendix entry on INT 21h Function 52h shows that the List of Lists actually begins at offset -12 (decimal) from the address returned in ES:BX. This wasn't important when we were using numeric offsets from the value Function 52h returns in ES:BX, but now that we're using a structure, we have to make sure we're really pointing at the beginning of the List of Lists.

This example demonstrates a fundamental problem with using data structures when working with undocumented DOS: structures are inflexible. The C compiler, seeing a reference such as doslist->vers.dos31.lastdrive, simply turns this into an offset into doslist. But these offsets are computed at compile time, not when the program is running, so they can't respond to run-time conditions such as different versions of an operating system.

Some of the simpler information-hiding features of C++ could be used to create a ListOfLists structure that responded to the DOS version number. When working with undocumented DOS data structures, programmers often wish they weren't so unruly, and one benefit of C++ is its ability to implement such "wishful thinking" by creating classes that manage and hide the complexity of underlying structures.

Most of us are working in C, however, not C++. Therefore, when using only one or two fields from an undocumented DOS data structure, and when placement of the fields within the structure differs from one DOS version to the next, it is best *not* to use data structures, but to compute offsets instead. Structures may be self-documenting, but they are also static. Remember the convoluted expression used earlier to extract the lastdrive field from the appropriate component of the union vers in struct ListOfLists? Note how much simpler it is when you use offsets:

```
if (_osmajor == 3 && _osminor == 0)
    lastdrv_ofs = 0x1B;
else if (_osmajor == 2)
    lastdrv_ofs = 0x10;
```

```
else
    lastdrv_ofs = 0x21;
lastdrv = doslist[lastdrv_ofs];

or:

lastdrv_ofs = (_osmajor == 3 && _osminor == 0) ? 0x1B :
              (_osmajor == 2) ?                  0x10 :
              /* otherwise */                    0x21 ;
lastdrv = doslist[lastdrv_ofs];
```

or the even more compact C expression, which also uses the C ?: ternary conditional operator in the next version of this utility:

```
/* LASTDRV6.C */

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

#ifdef __TURBOC__
#define ASM asm
#elif defined(_MSC_VER) && (_MSC_VER >= 600)
#define ASM _asm
#else
#error Requires inline assembler
#endif

unsigned _dos_lastdrive(void)
{
    char far *doslist;

    if (_osmajor < 2)
        return 0;

    ASM mov ah, 52h
    ASM int 21h
    ASM mov doslist+2, es
    ASM mov doslist, bx

    return doslist[(_osmajor == 3 && _osminor == 0) ? 0x1B :
                   (_osmajor == 2) ?                  0x10 :
                   /* otherwise */                    0x21];
}

main()
```

```
{
    unsigned lastdrive = _dos_lastdrive();
    if (lastdrive == 0xFF)
        return 0;
    fputs("LASTDRIVE=", stdout);
    putchar('A' - 1 + lastdrive);
    putchar('\n');
    return lastdrive;
}
```

The other item of interest in LASTDRV6.C is the use of in-line assembly within the function _dos_lastdrive(). In-line assembly language often seems like an invitation to produce *extremely* in-line code: C programmers encountering in-line assembly language for the first time seem to forget all about subroutines. Especially when working with the combination of undocumented DOS and in-line assembler, you should remember to use subroutines. But also remember our earlier warning to preserve the DI, SI, DS, SS, SP registers! The in-line assembler in _dos_lastdrive() only changes AX, BX, and ES, so we're okay here. The name was chosen to conform to the Microsoft C naming convention (_dos_getdrive(), _dos_setdrive(), etc.).

Undocumented DOS Calls From Turbo Pascal

Turbo Pascal programs that make undocumented DOS calls are similar to those that make documented calls, except that, as we saw with assembly language and C, such programs need to be especially aware of the version of MS-DOS under which they are running. The following program, LASTDRV2.PAS, uses the function DosVersion(), added in Turbo Pascal 5.0:

```
{ LASTDRV2.PAS }

program LastDrv;
uses dos;

var
    r : registers;
    lastdrv_ofs : Word;
    lastdrive : Word;
    vers : Word;

begin
    { determine offset of LASTDRIVE within DOS List of Lists }
```

```
    lastdrv_ofs := $21;
    vers := DosVersion;
    case Lo(vers) of
        0 : Halt(0); { DOS 1 }
        2 : lastdrv_ofs := $10;
        3 : if Hi(vers) = 0 then lastdrv_ofs := $1B;
    end;

    { Get pointer to DOS List of Lists }
    with r do begin
        ah := $52;
        es := 0; bx := 0;
        MsDos(r);
        if (es = 0) and (bx = 0) then
            Halt(0);
        lastdrive := Mem[es:bx+lastdrv_ofs];
    end;
    if lastdrive = $FF then
        Halt(0);

    { Print LASTDRIVE letter; return LASTDRIVE value }
    Writeln('LASTDRIVE=', Chr(Ord('A') - 1 + lastdrive));
    Halt(lastdrive);
end.
```

If you are working with a version of Turbo Pascal earlier than 5.0 and don't have the DosVersion() function, it is easy to write your own:

```
function DosVersion : Word;
var
    r : registers;
begin
    with r do begin
        ax := $3000;
        MsDos(r);
        DosVersion := ax;
    end;
end;
```

Note that LASTDRV2.PAS uses the predefined Turbo Pascal array Mem[] in order to peek at the DOS List of Lists. Mem[], MemW[], and MemL[] map onto the first megabyte of physical memory in the machine and are addressed with a segment:offset index, such as Mem[seg:ofs].

Rather than peek at a raw physical memory address with Mem[], we could use a data structure. Just as structures and unions can be used when making undocumented DOS calls from C, so variant records can be used from Turbo Pascal, as shown in LASTDRV3.PAS:

```
{ LASTDRV3.PAS }

program LastDrv;
uses dos;

type
    Dos20 = record
        numdrives : Byte;
        maxbytes : Word;
        first_diskbuff : Longint;
        nul : array [1..18] of Byte;
    end;

    Dos30 = record
        numblkdev : Byte;
        maxbytes : Word;
        first_diskbuff : Longint;
        currdir : Longint;
        lastdrive : Byte;
        stringarea : Longint;
        size_stringarea : Word;
        fcbtab : Longint;
        fcb_y : Word;
        nul : array [1..18] of Byte;
    end;

    Dos31 = record      { DOS 3.1 and higher }
        maxbytes : Word;
        diskbuff : Longint;
        currdir : Longint;
        fcb : Longint;
        numprotfcb : Word;
        numblkdev : Byte;
        lastdrive : Byte;
        nul : array [1..18] of Byte;
        numjoin : Word;
    end;

    ListOfLists = record
        shareretrycount : Word;
        shareretrydelay : Word;
```

```
        currdiskbuf : Longint;
        unreadcon : Word;
        mcb : Word;
        dpb : Longint;
        filetable : Longint;
        clock : Longint;
        con : Longint;
        case Word of
            20 : (dos20 : Dos20);
            30 : (dos30 : Dos30);
            31 : (dos31 : Dos31);
end;

var
    lastdrive : Word;

function GetLastDrive : Word;
var
    doslist : ^ListOfLists;
    r : registers;
    vers : Word;
begin
    { Get pointer to DOS List of Lists }
    with r do begin
        ah := $52;
        es := 0; bx := 0;
        MsDos(r);
        if (es = 0) and (bx = 0) then begin
            GetLastDrive := 0;
            Exit;
        end;
        doslist := Ptr(es, bx - 12);
    end;
    { LASTDRIVE offset depends on DOS version }
    GetLastDrive := doslist^.dos31.lastdrive;
    vers := DosVersion;
    case Lo(vers) of
        0 : GetLastDrive := 0; { DOS 1 }
        2 : GetLastDrive := doslist^.dos20.numdrives;
        3 : if Hi(vers) = 0 then
                GetLastDrive := doslist^.dos30.lastdrive;
    end;
end;

begin
    lastdrive := GetLastDrive;
    if lastdrive = 0 then
```

```
        Halt(0);
    Writeln('LASTDRIVE=', Chr(Ord('A') - 1 + lastdrive));
    Halt(lastdrive);
end.
```

LASTDRV3.PAS has nice self-documenting structures but doesn't adjust itself to the DOS version number as well as LASTDRV2.PAS, which simply used numeric offsets. This is the same tradeoff we saw when using the C programming language.

Undocumented DOS Calls From BASIC

The first BASIC version of the LASTDRV utility, which used only documented DOS calls, required a DOSEXIT() subroutine in order to return an exit code to MS-DOS. The second BASIC version of LASTDRV, which uses the undocumented DOS List of Lists, also needs a DOSVERSION() function so that it can determine the offset of LASTDRIVE within the DOS List of Lists:

```
REM LASTDRV2.BAS
REM $INCLUDE: 'QB.BI'

DEF FNHI (x) = x \ &H100
DEF FNLO (x) = x AND &HFF

FUNCTION DOSVERSION
    DIM Regs AS RegType
    Regs.ax = &H3000
    CALL INTERRUPT(&H21, Regs, Regs)
    DOSVERSION = Regs.ax
END FUNCTION

SUB DOSEXIT(errorlevel)
    CLOSE
    DIM Regs AS RegType
    Regs.ax = &H4C00 + errorlevel
    CALL INTERRUPT(&H21, Regs, Regs)
END SUB

REM based on DOS version number, find offset of LASTDRIVE
lastdrvofs = &H21
vers = DOSVERSION
IF FNLO(vers) < 3 THEN DOSEXIT(0)
IF (FNLO(vers) = 3) AND (FNHI(vers) = 0) THEN lastdrvofs = &H1B
```

```
REM get address of DOS List of Lists
DIM Regs AS RegTypeX
Regs.ax = &H5200
Regs.es = 0
Regs.bx = 0
REM to use current value of DS, set to -1
Regs.ds = -1
CALL INTERRUPTX(&H21, Regs, Regs)
IF (Regs.es = 0) AND (Regs.bx = 0) THEN DOSEXIT(0)

REM peek at LASTDRIVE field within DOS List Of Lists
DEF SEG = Regs.es
lastdrv = PEEK(Regs.bx + lastdrvofs)
IF lastdrv = &HFF THEN DOSEXIT(0)

REM print LASTDRIVE letter, return LASTDRIVE number
PRINT "LASTDRIVE="; CHR$(ASC("A") - 1 + lastdrv)
CALL DOSEXIT(lastdrv)
END
```

Once INT 21h Function 52h returns the address of the List of Lists in the ES:BX register pair, DEF SEG and PEEK() are used to read the LASTDRIVE field. Instructions for compiling this code into a stand-alone executable can be found in the earlier section on "DOS calls from BASIC."

When Not to Use Undocumented Features

You might think that the last few sections descended into the very depths of DOS simply to bring back a piece of information that was readily available all the while using DOS's well-documented function interface. This could be compared to an American who learns Japanese and then uses his newly acquired skill only to watch American movies dubbed into Japanese.

This provides us with a fine example of when *not* to use undocumented DOS. If there is a way to perform an operation using the documented DOS programmer's interface, use it. In fact, *go out of your way* to use the documented interfaces. If there is a seemingly convenient way to accomplish some task using the undocumented calls described in this book, and a less convenient way involving only documented calls, use the documented calls. (You'll see a good example of this in chapter 3 where we discuss the temptation to use INT 29h.)

The "Mount Everest" approach to programming—the desire to use a function, simply because it is there—is wonderful when you are experimenting with a new

operating system, but it has no place in commercial software. One of our worries in producing this book was that it might encourage the over-use of undocumented DOS. Please don't use undocumented DOS when documented DOS will do.

Having said all this, though, let's remember that lots of successful commercial software on the PC uses undocumented DOS features. Certain things can't be done using only the documented interfaces. This is somewhat analogous to the situation with DOS calls, BIOS calls, and direct hardware access: clearly direct hardware access should be used only as a last resort, but almost all successful PC software does some direct hardware access!

Verifying Undocumented DOS

Actually, there is one good reason for using undocumented DOS when there is equivalent documented functionality. It would be nice to have a way to perform a baseline validation of the usability of undocumented DOS in any given environment. Obviously, the best way to validate a value computed using undocumented DOS is to compare it to a known value with which it should be equivalent.

It seems we can't double check the results of undocumented DOS against documented DOS because, if we could, we would be using documented DOS in the first place! To check that doslist->lastdrive really is equal to setdisk(getdisk()), for example, seems pointless. But what if you were interested in some value other than LASTDRIVE from the DOS List of Lists? Then, successfully comparing doslist->lastdrive against a known value might give your program enough confidence to proceed using undocumented DOS, whereas a mismatch might indicate that something is very wrong.

Therefore, you might want to incorporate something similar to the following function in your programs:

```
typedef enum { FALSE, TRUE } BOOL;

/* one possible way of veryifying undocumented DOS */
BOOL undoc_dos_okay(void)
{
    char far *doslist;

    /* get offset for LASTDRIVE within DOS List of Lists */
    unsigned lastdrv_ofs = 0x21;
    if (_osmajor==2) lastdrv_ofs = 0x10;
```

```
    else if ((_osmajor==3) && (_osmajor==0)) lastdrv_ofs = 0x1b;

    /* Get DOS Lists of Lists */
    ASM mov ah, 52h
    ASM xor bx, bx
    ASM mov es, bx
    ASM int 21h
    ASM mov doslist, bx
    ASM mov doslist+2, es

    if (! doslist)
        return FALSE;

    /* use documented DOS to verify results */
#ifdef __TURBOC__
    return (setdisk(getdisk()) == doslist[lastdrv_ofs]);
#else
    {
        unsigned drive;
        unsigned lastdrive;
        _dos_getdrive(&drive);
        _dos_setdrive(drive, &lastdrive);
        return (lastdrive == doslist[lastdrv_ofs]);
    }
#endif
}
```

If undoc_dos_okay() returns TRUE in, say, DOS 7.8, it is no guarantee that all code that employs undocumented DOS will work. However, if undoc_dos_okay() returns FALSE, there's a good chance that your code will need to be fixed before it will run properly. For example, undoc_dos_okay() fails in the OS/2 1.10 DOS box, but it succeeds in the vastly improved multiple DOS boxes of OS/2 2.0.

An Important Special Case: Novell NetWare

We noted earlier that Quarterdeck's LASTDRIV.COM utility uses the undocumented rather than the documented technique for retrieving the value of LASTDRIVE. One reason for this seemingly outrageous flouting of the normal rules of good behavior is that LASTDRIV.COM can also be used to *change* the value of LASTDRIVE, dynamically adding or subtracting drives from DOS's internal table. That is precisely the kind of operation which requires undocumented DOS.

But there's an additional reason why Quarterdeck's LASTDRIV.COM uses undocumented DOS for getting LASTDRIVE: the documented method is actually *less* reliable than the undocumented method! On any of the many PCs that are Novell NetWare workstations, INT 21h Function 0Eh doesn't return the value of LASTDRIVE; it returns the number 32, corresponding to the number of possible workstation drive mappings (drive letters A through Z, plus temporary drives with the silly names [, \,], ^, _, and '). Thus, under NetWare, the versions of LASTDRV which use undocumented DOS display correct values for LASTDRIVE, whereas the supposedly "well-behaved" version of LASTDRV that uses only documented DOS always prints out the following display:

```
C>lastdrv
LASTDRIVE='
```

Likewise, under NetWare our carefully written undoc_dos_okay() function returns FALSE! This happens not because undocumented DOS is "broken" but because documented Function 0Eh is returning a strange value.

It is important to look into this. Novell is by far the largest supplier of PC local area network (LAN) software; its share of the PC LAN software market is twice that of even IBM. Therefore, if the version of LASTDRV that uses documented DOS doesn't work under Novell, it essentially doesn't work!

In any case, this gives us an excuse to look into what it means for a program to "hook DOS." How does the NetWare shell running on a workstation change DOS so that Function 0E returns 32 instead of the value of LASTDRIVE? Easy: it "hooks DOS." That is, the NetWare shell inspects every INT 21h function request before DOS itself sees it, and the shell decides whether to pass that function request along to DOS or to pass the request over the network to another machine, the server (which is not even a DOS machine!). Finally, even if the shell does decide to pass the INT 21h function request along to DOS, it gets to modify any registers before returning control to the application (such as LASTDRV) that called INT 21h in the first place.

In order to hook DOS, all a program has to do is get the address of the current interrupt handler for INT 21h and then install its own handler for INT 21h. There's nothing difficult or undocumented about this capability: it's built right into DOS itself and is one of the key facilities that makes DOS so extensible.

In fact, it's so simple that we can simulate NetWare's handling of Function 0Eh, and provide a realistic example of hooking DOS, with just a few lines of

code. Generally programs that hook DOS (like Novell's ANET3) are memory-resident. However, building a TSR would complicate this discussion unnecessarily, so the following program, FUNC0E32, instead acts as a "shell" around another program. For example:

```
C:\UNDOC>func0e32 lastdrv.exe
LASTDRIVE='

10 DOS calls
1 changed
```

Note, however, that although the documented version of LASTDRV is fooled by FUNC0E32, the version that uses undocumented DOS isn't:

```
C:\UNDOC>func0e32 lastdrv.exe
LASTDRIVE=E

10 DOS calls
1 changed
```

FUNC0E32 consists of two functions: The function dos() is our INT 21h handler. Each time INT 21h is invoked, we want dos() to get control. The function changes the value Function 0Eh returns in AL to 32, and also keeps count of how many INT 21h calls it has seen and how many it has changed. The function main() installs dos() as the INT 21h handler, spawns the program named on the command line, and then restores the original INT 21h handler (which may be DOS itself, but which, on most PCs, will be some other program that had earlier hooked DOS, such as NetWare, CED, the Epsilon text editor, etc.):

```
/*
FUNC0E32.C -- take over INT 21h Function 0Eh; return 32 in AL
*/

#include <stdlib.h>
#include <stdio.h>
#include <process.h>
#include <dos.h>

#pragma pack(1)

typedef struct {
```

```
#ifdef __TURBOC__
    unsigned bp,di,si,ds,es,dx,cx,bx,ax;
#else
    unsigned es,ds,di,si,bp,sp,bx,dx,cx,ax;     /* same as PUSHA */
#endif
    unsigned ip,cs,flags;
    } REG_PARAMS;

void interrupt far dos(REG_PARAMS r);

void (interrupt far *old)();
unsigned long calls = 0;
unsigned long changed = 0;

void fail(char *s) { puts(s); exit(1); }

main(int argc, char *argv[])
{
    if (argc < 2)
        fail("usage: func0e32 [program name] <args...>");

    /* hook INT 21 */
    old = _dos_getvect(0x21);
    _dos_setvect(0x21, dos);

    /* run command */
    spawnvp(P_WAIT, argv[1], &argv[1]);

    /* unhook INT 21h */
    _dos_setvect(0x21, old);
    printf("\n%lu DOS calls\n", calls);
    printf("%lu changed\n", changed);
}

void interrupt far dos(REG_PARAMS r)
{
    calls++;
    if ((r.ax >> 8) == 0x0E)
    {
        (*old)();
        r.ax = 0x0E00 + 32;
        changed++;
    }
    else
        _chain_intr(old);
}
```

This code is important, not only to illustrate how it is perfectly legal (and completely documented!) for a company like Novell to change the return value from a DOS function but also as an example of how to hook a DOS interrupt like INT 21h. Some undocumented DOS functions are not for you to call, but for you to *implement* so that DOS can call them ("don't call us, we'll call you"). Such functions are indicated in Appendix A with the phrase "Called with" rather than "Call with." For example, the DOS network redirector (which, incidentally, Novell does not use), is one such set of "call back" functions (actually, subfunctions to INT 21h Function 11h).

Novell's altering of Function 0Eh was probably a mistake: in fact, Novell provides an alternate function, INT 21h Function DBh, which *does* return the correct value of LASTDRIVE. This is not part of undocumented DOS, so it does not appear in Appendix A; however, it is documented in the "Interrupt List" on the accompanying disk.

To be compatible with NetWare, we need to change all of our validity checking. Instead of asserting that undocumented DOS is unusable simply because doslist->lastdrive != setdisk(getdisk()), we now perform a slightly more complicated test, as shown in the following improved version of undoc_dos_okay():

```
BOOL netware(void);
unsigned lastdrv_netware(void);

BOOL undoc_dos_okay(void)
{
    char far *doslist;
    unsigned lastdrv_doc;
    unsigned drive;

    /* get offset for LASTDRIVE within DOS List of Lists */
    unsigned lastdrv_ofs = 0x21;
    if (_osmajor==2) lastdrv_ofs = 0x10;
    else if ((_osmajor==3) && (_osmajor==0)) lastdrv_ofs = 0x1b;

    /* Get DOS Lists of Lists */
    ASM mov ah, 52h
    ASM xor bx, bx
    ASM mov es, bx
    ASM int 21h
    ASM mov doslist, bx
    ASM mov doslist+2, es
```

```
    if (! doslist)
        return FALSE;

    /* use documented DOS to verify results */
#ifdef __TURBOC__
    lastdrv_doc = setdisk(getdisk());
#else
    _dos_getdrive(&drive);
    _dos_setdrive(drive, &lastdrv_doc);
#endif
    if (doslist[lastdrv_ofs] == lastdrv_doc)
        return TRUE;
    else if (netware())
    {
        if (lastdrv_doc != 32)
            puts("NetWare Function 0Eh looks strange");
        return (doslist[lastdrv_ofs] == lastdrv_netware());

    return FALSE;
}

/*
    Novell Return Shell Version function (INT 21h AH=EAh AL=01h)
    see "Interrupt List" on accompanying disk; also see Barry
    Nance, Networking Programming in C, pp. 117, 341-2. Could also
    test for presence of Novell IPX with INT 2Fh AX=7A00h.
*/
BOOL netware(void)
{
    char buf[40];
    char far *fp = buf;
    ASM push di
    ASM mov ax, 0EA01h
    ASM mov bx, 0
    ASM les di, fp
    ASM int 21h
    ASM xor al, al
    ASM mov ax, bx
    /* if BX still 0, then NetWare not present; return in AX */
    ASM pop di
}

/*
    Novell Get Number of Local Drives function (INT 21h AH=DBh)
    see "Interrupt List" on accompanying disk
*/
unsigned lastdrv_netware(void)
```

```
{
    ASM mov ah, 0DBh
    ASM int 21h
    /* AL now holds number of "local drives" (i.e., LASTDRIVE) */
    ASM xor ah, ah
    /* unsigned returned in AX */
}
```

Undocumented DOS Calls from Protected Mode

It is testimony to the great diversity of MS-DOS that we are still not finished with our catalog of the basic ways to make undocumented DOS calls.

We have often referred to changes in the internal structure of DOS from one version to the next. These versions of DOS must be taken to include, not only DOS 3.1, 4.0, and so on, but also the most important extensions of DOS: Microsoft Windows 3.0, the OS/2 compatibility box, the DOS Protected-Mode Interface (DPMI), and protected-mode DOS extenders such as Phar Lap's 386|DOS-Extender (incorporated in such products as AutoCAD/386, IBM Interleaf Publisher, and Mathematica) and Rational Systems's DOS/16M (incorporated in Lotus 1-2-3 Release 3, for example).

A thorough discussion of the world of protected-mode DOS may be found in the book *Extending DOS*, edited by Ray Duncan (Reading, MA: Addison-Wesley, 1990). Here, we need to look quickly at how running in protected mode alters the way undocumented DOS calls are made.

DOS extenders break the DOS 640KB barrier by running your application in the protected mode of the 80286, 80386, and 80486 processors. Your application continues to use the services of MS-DOS, which is running in real mode in the lower 640KB. DOS extenders make access to DOS as transparent as possible. For example, a protected-mode application allocates memory using INT 21h Function 48h, just as do real-mode DOS applications. The difference is that, under a DOS extender, INT 21h Function 48h can allocate multiple megabytes of extended memory.

Because it is not limited to one megabyte of immediately addressable memory, protected mode uses a fundamentally different addressing scheme than real mode. DOS extenders transparently handle almost all of these differences for you: your application makes normal DOS calls, and the DOS extender takes care of the rest.

Notice that we just said "normal DOS calls." What happens when you take an application that makes undocumented DOS calls and port it to a protected-mode DOS extender? According to the Phar Lap manual, "386 | DOS-Extender extends all of the documented MS-DOS system calls, and most of the BIOS system calls, so that they can be made directly from protected mode. However, some programs also need access to undocumented MS-DOS functions . . . If the system call uses segment registers, then additional processing is required by the protected mode program." What does this additional processing look like?

All DOS extenders provide a small set of services for making real-mode software interrupts from protected mode. Usually you don't need to use this service, because the DOS extender already provides the software interrupt in protected mode. Undocumented DOS, however, is a perfect example of a case in which you need to use these special services.

In the remainder of this section, we will look at two final examples of the LASTDRV utility. Naturally, tiny programs like this are not likely ever to use a DOS extender. But large programs that are likely candidates for using protected mode may well contain undocumented DOS calls, so it is important to know how to get these working in this new and important environment.

386|DOS-Extender

Our first example uses Phar Lap 386 | DOS-Extender, which runs 32-bit applications under MS-DOS on 80386 and 80486 processors. Watcom C/386, one of several 32-bit C compilers now available for MS-DOS, is used as well. Note that 386 | DOS-Extender requires DOS 3.0 or greater, which simplifies our DOS version checking:

```
/* LD386.C -- undocumented DOS call from 386|DOS-Extender */

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

#ifndef __WATCOMC__
#error This program requires Watcom C/386
```

```
#endif

typedef struct {
    unsigned short intno, ds, es, fs, gs;
    unsigned eax, edx;
    } RMODE_PARAM_BLOCK;

main()
{
    RMODE_PARAM_BLOCK rpb;
    union REGS r;
    struct SREGS s;
    char far *doslist;
    unsigned lastdrv;

    /* load real-mode param block for INT 21h AH=52h */
    memset(&rpb, 0, sizeof(RMODE_PARAM_BLOCK));  /* zero it out */
    rpb.intno = 0x21;
    rpb.eax = 0x5200;

    /* call 386|DOS-Extender service to "Issue Real Mode Interrupt,
        Registers Specified" (INT 21h AX=2511h). */
    r.x.eax = 0x2511;
    r.x.edx = &rpb;
    segread(&s);
    int386x(0x21, &r, &r, &s);

    /* use 386|DOS-Extender selector 34h (writeable data segment that
        maps the entire first megabyte of memory used by MS-DOS) */
    doslist = MK_FP(0x34, (rpb.es << 4) + r.x.ebx);

    /* we now have protected-mode ptr to DOS List Of Lists --
        use normally */
    lastdrv = doslist[_osmajor==3 && _osminor==0 ? 0x1B : 0x21];
    fputs("LASTDRIVE=", stdout);
    putchar('A' - 1 + lastdrv);
    putchar('\n');
    return lastdrv;
}
```

This example can be compiled, linked, and run with the following DOS command lines (WCL386 is the Watcom C driver program, and RUN386 is the Phar Lap DOS extender):

```
wcl386 lastdrv
run386 lastdrv
```

In this example, we call the service to issue a real-mode interrupt, specifying our INT 21h Function 52h call inside the "real-mode parameter block." Thus, we invoke Phar Lap Function 2511h, and ask *it* to invoke DOS Function 52h in real mode. This sort of indirect call is only necessary for those weird calls (like undocumented DOS) not transparently supported in protected mode. This indirect method of making the undocumented DOS call is typical of all DOS extenders. When we return from the real-mode interrupt, the ES:EBX register pair contains an address such as 028E:00000026 (offsets are four bytes wide in true 80386 code). This is a real-mode address, and has no real meaning in protected mode. In order to use this address, we need to turn it into a protected-mode pointer. In 386 | DOS-Extender selector 34h is a writeable data segment which maps the entire first megabyte of memory (the entire DOS machine fits in one small portion of one 32-bit protected-mode selector!). We form a six-byte protected-mode far pointer using the Watcom C MK_FP() macro, and then use it as we would normally.

DPMI

Our second example doesn't actually use a DOS extender per se, but the DOS Protected-Mode Interface (DPMI), a specification drawn up by a committee comprised of Microsoft, Intel, IBM, Lotus, Phar Lap, Rational Systems, Borland, Quarterdeck, and other companies. DPMI describes a set of services which can be called from protected mode using INT 31h. Providers of these services, such as Microsoft Windows 3.0, are known as *DPMI servers*, whereas users of these services, such as protected-mode DOS extenders, are *DPMI clients*. A DPMI server can be built in many different environments, including OS/2 2.0 or even UNIX, offering the possibility (when all this is actually implemented) of running DOS extended applications in many different (even non-DOS) environments. Maybe one day INT 21h and ROM BIOS calls *will* be available on non-Intel architectures!

Unlike the expanded memory (EMS) or extended memory (XMS) specifications, DPMI is not intended to be used in application programs. DPMI is really meant for use by a DOS extender which, in turn, provides services to application programs. It is nonetheless instructive to examine a DPMI version of LASTDRV. Many Windows 3.0 programmers, who need to access real-mode drivers and TSRs from Windows enhanced mode, have had no choice but to learn about DPMI.

The following sample program uses Microsoft C 6.0. The program starts up in real mode, and makes an INT 2Fh AX=1687h call to check for the presence of DPMI (note that INT 2Fh AH=16h and AH=17h generally is used as a Microsoft Windows interface for non-Windows applications; for more information, see the on-disk "Interrupt List"). If DPMI is present (for example, if running in a DOS box in Windows 3.0 enhanced mode), the address of the DPMI "protected-mode switch entry point" is returned in ES:DI. This is a function which switches the program into protected mode. Because this program starts off in real mode and then switches into protected mode, all the segment registers change in mid-stream! If we want to use the C standard library, then, the program *must* be compiled with small model:

```
/*
LDDPMI.C -- undocumented DOS call from DPMI
see updated version on disk in CHAP2\LDDPMI.C
sample output:
    in protected mode
    Real mode DOS List Of Lists = 028E:0026
    Protected DOS List Of Lists = 00AD:0026
    LASTDRIVE=E

cl -AS lddpmi.c
*/

#ifndef M_I86SM
#error Requires Microsoft C small model
#endif

#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <dos.h>

#define ABSADDR(seg, ofs) \
    ((((unsigned long) seg) << 4) + ((ofs) & 0xFFFF))

#pragma pack(1)

typedef struct {
    unsigned long edi, esi, ebp, reserved, ebx, edx, ecx, eax;
    unsigned flags, es, ds, fs, gs, ip, cs, sp, ss;
    } RMODE_CALL;
```

```
typedef struct {
    unsigned accessed   : 1;
    unsigned read_write : 1;
    unsigned conf_exp   : 1;
    unsigned code       : 1;
    unsigned xsystem    : 1;
    unsigned dpl        : 2;
    unsigned present    : 1;
    } ACCESS;

typedef struct {
    unsigned        limit;
    unsigned        addr_lo;
    unsigned char   addr_hi;
    ACCESS          access;
    unsigned        reserved;
    } DESCRIPTOR;

typedef enum { FALSE, TRUE } BOOL;

BOOL dpmi_rmode_intr(unsigned intno, unsigned flags,
    unsigned copywords, RMODE_CALL far *rmode_call);

void dos_exit(unsigned err)
{
    _asm mov al, err
    _asm mov ah, 04ch
    _asm int 21h
}

void pmode_putchar(int c)   //call real-mode INT21 AH=2 from pmode
{
    static RMODE_CALL r;
    static RMODE_CALL *pr = (void *) 0;
    if (! pr)
    {   //just do one time
        pr = &r;
        memset(pr, 0, sizeof(RMODE_CALL));
        r.eax = 0x0200;
    }
    r.edx = c;
    dpmi_rmode_intr(0x21, 0, 0, pr);
}

void pmode_puts(char *s)
{
    while (*s)
```

```
    {
        pmode_putchar(*s);
        s++;
    }
    pmode_putchar(0x0d);
    pmode_putchar(0x0a);
}

void cdecl pmode_printf(const char *fmt, ...)
{
    char buf[128], *s=buf;
    va_list marker;   //use ANSI C stdarg facility
    va_start(marker, fmt);
    vsprintf(buf, fmt, marker);
    va_end(marker);
    while (*s)
        pmode_putchar(*s++);
}
void fail(char *s)       { puts(s); exit(1); }
void pmode_fail(char *s) { pmode_puts(s); dos_exit(1); }

/* Determines if DPMI is present and, if so, switches into
   protected mode */
BOOL dpmi_init(void)
{
    void (far *dpmi)();
    unsigned hostdata_seg, hostdata_para, dpmi_flags;
    _asm {
        mov ax, 1687h           ; test for DPMI presence
        int 2Fh
        and ax, ax
        jnz nodpmi              ; if (AX == 0) DPMI is present
        mov dpmi_flags, bx
        mov hostdata_para, si   ; paras for DPMI host private data
        mov dpmi, di
        mov dpmi+2, es          ; DPMI protected-mode switch entry point
        jmp short gotdpmi
        }
nodpmi:
    return FALSE;
gotdpmi:
    if (_dos_allocmem(hostdata_para, &hostdata_seg) != 0)
        pmode_fail("can't allocate memory");

    /* enter protected mode */
    _asm {
        mov ax, hostdata_seg
```

```
        mov es, ax
        mov ax, dpmi_flags
        }
    (*dpmi)();

    return TRUE;
}

/* Performs a real-mode interrupt from protected mode */
BOOL dpmi_rmode_intr(unsigned intno, unsigned flags,
    unsigned copywords, RMODE_CALL far *rmode_call)
{
    if (flags) intno |= 0x100;
    _asm {
        push di
        push bx
        push cx

        mov ax, 0300h       ; simulate real-mode interrupt
        mov bx, intno       ; interrupt number, flags
        mov cx, copywords;  ; words to copy from pmode to rmode stack
        les di, rmode_call  ; ES:DI = address of rmode call struct
        int 31h             ; call DPMI
        jc error
        mov ax, 1           ; return TRUE
        jmp short done
error:  mov ax, 0           ; return FALSE
done:   pop cx
        pop bx
        pop di
        }
}

/* Allocates a single protected-mode LDT selector */
unsigned dpmi_sel(void)
{
    _asm {
        mov ax, 0           ; Allocate LDT Descriptors
        mov cx, 1           ; allocate just one
        int 31h             ; call DPMI
        jc err
        jmp short done      ; AX holds new LDT selector
err:    mov ax, 0           ; failed
done:   }
}

BOOL dpmi_set_descriptor(unsigned pmodesel, DESCRIPTOR far *d)
```

```
{
    _asm {
        push di
        push bx
        mov ax, 000ch       ; Set Descriptor
        mov bx, pmodesel    ; protected mode selector
        les di, d           ; descriptor
        int 31h             ; call DPMI
        jc error
        mov ax, 1           ; return TRUE
        jmp short done
error:  mov ax, 0           ; return FALSE
done:   pop di
        pop bx
        }
}

BOOL dpmi_sel_free(unsigned pmodesel)
{
    _asm {
        mov ax, 0001h       ; Free LDT Descriptor
        mov bx, pmodesel    ; selector to free
        int 31h             ; call DPMI
        jc error
        mov ax, 1           ; return TRUE
        jmp short done
error:  mov ax, 0           ; return FALSE
done:   }
}

main(int argc, char *argv[])
{
    DESCRIPTOR d;
    RMODE_CALL r;
    void far *fp;
    char far *doslist = (char far *) 0;
    unsigned long addr;
    unsigned pmodesel;
    unsigned offset, lastdrv_ofs, lastdrv;

    /*
        Determine if DPMI present and, if so, switch
        to protected mode
    */
    if (dpmi_init())
        pmode_puts("now in protected mode");
    else
```

```
    fail("DPMI not present");
/*
    Call INT 21h AH=52h (Get DOS List Of Lists)
*/
memset(&r, 0, sizeof(RMODE_CALL));
r.eax = 0x5200;
if (! dpmi_rmode_intr(0x21, 0, 0, &r))
    pmode_fail("DPMI rmode intr failed");
FP_SEG(doslist) = r.es;
FP_OFF(doslist) = r.ebx;
pmode_printf("Real mode DOS List Of Lists = %Fp\r\n", doslist);

/* doslist now holds a real-mode address: in order to address it
   in protected mode, allocate an LDT descriptor and set its
   contents; when done, deallocate the LDT descriptor
*/
if (! (pmodesel = dpmi_sel()))
    pmode_fail("DPMI can't alloc pmode selector");
d.limit = 0xFFFF;
addr = ABSADDR(r.es, 0);
d.addr_lo = addr & 0xFFFF;
d.addr_hi = addr >> 16;
d.access.accessed = 0;      /* never been used */
d.access.read_write = 1;    /* read-write */
d.access.conf_exp = 0;      /* not a stack */
d.access.code = 0;          /* data */
d.access.xsystem = 1;       /* not system descriptor */
fp = (void far *) main;
d.access.dpl = FP_SEG(fp) & 3;  /* protection level */
d.access.present = 1;       /* it's present in memory */
d.reserved = 0;
if (! dpmi_set_descriptor(pmodesel, &d))
    pmode_fail("DPMI can't set descriptor");

FP_SEG(doslist) = pmodesel; /* convert to protected-mode address */
FP_OFF(doslist) = r.ebx;
pmode_printf("Protected mode DOS List Of Lists = %Fp\r\n", doslist);

/* now have protected-mode selector to DOS List of Lists */
/* Get LASTDRIVE number, print LASTDRIVE letter */
lastdrv = doslist[_osmajor==3 && _osminor==0 ? 0x1b : 0x21];
pmode_printf("LASTDRIVE=%c\r\n", 'A' - 1 + lastdrv);

if (! dpmi_sel_free(pmodesel))
    pmode_fail("DPMI can't free selector");

/* in protected mode, flush output and quit */
dos_exit(0);
```

```
dpmifail:
    pmode_fail("DPMI failure");
}
```

There is a lot of code here, but the workings of LDDPMI are actually fairly simple: in order to call INT 21h Function 52h from protected mode, we must do so indirectly via INT 31h Function 0300h (see the function dpmi_rmode_intr). What we get back, of course, is the *real-mode* address is the DOS List of Lists, which we must convert into a protected-mode address. We therefore allocate a descriptor from the Local Descriptor Table (LDT; see the function dpmi_sel), and set its base address. An image of the descriptor is created in main; it is placed in the LDT using INT 31h Function 000Ch (see dpmi_set_descriptor). We now have a protected-mode pointer to the DOS List of Lists, which we can use in the usual way. Before exiting, we free up the descriptor (see dpmi_sel_free).

Sheesh! All this mixing of real and protected modes in the same program is pretty hair-raising, but the next few years of MS-DOS will probably see more and more of this sort of code.

In addition to the restriction to small-model, another restriction is that, once you enter protected mode, you can't debug this thing using a real-mode debugger like CodeView: stepping over the (*dpmi)() call in dpmit_init() hangs the machine. These restrictions illustrate why, even with the availability of DPMI, you probably want to use a commercial DOS extender rather than "roll your own."

Because this program switches in midstream from real to protected mode, pointers that are going to be used in real mode can't be initialized on the real-mode side of the fence. Instead, the program must not assign the value to fpr until *after* it has switched into protected mode.

Note that, after switching into protected mode, LDDPMI uses functions such as pmode_printf() rather than plain old printf(). The reason is that, somewhere in the depths of the C run-time library, functions such as printf() eventually make INT 21h calls. But recall that we are now running in protected mode. Most DPMI servers will in fact provide protected-mode INT 21h services (the Windows 3.x DOS extender does, for example), but that is a facility provided by the DPMI server, not by DPMI itself. Therefore, a few routines have been cobbled together in LDDPMI so we can do output.

With the comparative intricacies of this program, it is a relief that at least the DOS version checking is greatly simplified: we don't ever have to worry about a DPMI server running under DOS 2.x!

Chapter 3

MS-DOS Resource Management: Memory, Processes, Devices

Jim Kyle

Resource management is the primary task of any operating system, and MS-DOS is no exception. This chapter concentrates on such facets of resource management as device drivers, DOS memory allocation, and process management.

Throughout the discussion, sample code fragments and programs are used; the conclusion brings everything together in a utility that lets you install a device driver from the DOS command line, without requiring that you edit your CONFIG.SYS file or reboot the system.

The earliest operating systems, in the dim prehistory of mainframe days, managed resources by default: only one process could be loaded into the machine at a time, and that process had full access to all resources.

Operating systems evolved, though, and it became possible to load several processes at the same time. Any true operating system must, in fact, contain at least two processes: the *supervisor* or *system* program (often called the *kernel*), and the real user program. As soon as there is more than one process it becomes necessary to manage memory and devices so that no process intrudes on another.

Memory Management

MS-DOS allows programs to allocate, free, and resize memory through three documented functions (INT 21h, Functions 48h, 49h, and 4Ah), but the actions of the DOS memory manager itself are not officially documented. This section describes how memory is organized to make these functions possible.

The memory management scheme used by MS-DOS divides the first megabyte of the system's memory into contiguous blocks, each of which has a Memory Control Block (MCB) as its first paragraph. Each MCB provides enough information to get to the next MCB. It is important to note that this chain is *not* a linked list but a contiguous block of memory: the size of one block is added onto its starting address to get to the next block. This structure is referred to in official documentation (and in Ray Duncan's better-than-official *Advanced MS-DOS Programming*) as the *memory arena*, and the MCB that begins each block is referred to as an *arena header*.

The initial memory arena structure is built at system boot time, just after the parsing of CONFIG.SYS directives. This structure omits memory below the DOS data segment, because all RAM in that area was assigned earlier in the boot-up procedure and is not subject to reallocation.

Memory Control Blocks

Each block of memory begins with an MCB, which is a single paragraph. That is, the MCB is 16 bytes long and begins at an address that is an exact multiple of 16. The memory block itself also is always an exact number of paragraphs in length. This paragraph alignment makes it possible to refer to a memory block using a 16-bit segment address rather than a full 20-bit address.

When a program requests a block of memory with INT 21h Function 48h, DOS must find the number of requested paragraphs, plus one more for the MCB. Assuming that a block of memory is available, DOS sets up its first paragraph as an MCB and hands the segment address of the *second* paragraph back to the program. Let's say you've made this call:

```
mov ah, 48h          ; Allocate Memory Block
mov bx, 1            ; one paragraph
int 21h
jc fail
; AX now holds initial segment of allocated block
```

Let's say AX now holds the value 1234h. This means that an MCB is located at 1233h. What does this MCB look like?

The first byte of every MCB except the last one in the chain is 4Ch (the ASCII code for 'M'); the last MCB's first byte is instead 5Ah ('Z'). It may be only coincidental that these two letters are the initials of the principal architect of the DOS memory manager, Mark Zbikowski. In our example, this field could either be 'M' or 'Z', though 'M' is far more likely.

Following this tag byte is a 16-bit value (in Intel low-high format) that identifies the "owner" of the MCB. These 16 bits will be 0000 if the memory block is available for use. Otherwise, they will contain the ID number of the process to which the block has been allocated (the "owner" process). This information is used to locate free blocks (owner is zero) and to release allocated blocks when a process terminates. This ID number is the Program Segment Prefix (PSP; see below) of the owner. In our example, this field would hold the PSP of whatever program called INT 21h Function 48h.

Following the "owner" word is another word giving the size in paragraphs of the memory block controlled by this MCB. In our example, this field will be set to 1, indicating that the MCB at 1233h controls only the next paragraph at 1234h. In other words, this size value does *not* include the paragraph taken by the MCB itself; consequently, it's possible to have a valid MCB that shows a free block with size equal to zero. This happens when all but one paragraph of a previously free block is allocated, and it is, in fact, not unusual.

The three bytes following the "size" word are unused in all versions of MS-DOS to date. In versions 2 and 3, all remaining bytes of the MCB were unused, but in DOS 4.x the final 8 bytes of the MCB may contain the filename of the owning program. The name is included only in the MCB that controls the memory used by the program's PSP; otherwise, the final 8 bytes are ignored.

To find the next MCB in memory (remember, it's not really a linked list), you start with the MCB's own segment address, add 1 to it to get the segment address of the RAM it controls, then add to that the size from the word at byte 3 of the MCB. The result is the segment address of the next MCB. In our example, the next MCB is at 1235h. If the byte at 1235:0000 is anything other than 'M' or 'Z', the MCB chain has been corrupted and continued operation is not possible.

The first MCB is always the one that controls DOS's own data segment. Its "owner" word is usually 0008h, for reasons that are not fully understood. It corresponds to the memory allocated based on commands given in CONFIG.SYS.

The final MCB, identified by 'Z' as its first byte, will in a normal 640KB system generate a "next-MCB" address of 0A000h, although that address should not be used, because the 'Z' code indicates no "next MCB" exists.

In DOS 4.x and higher, the DOS data segment memory block is subdivided into "subsegments"; each subsegment has its own variant of the standard MCB. However, the 'M'-coded MCB for the data segment includes the entire area, so you don't need to trace the subsegments when going through the MCB chain.

The DOS 4.x subsegments follow a format similar to, but not identical with, the MCB layout: the first byte is a letter indicating usage, but the word at byte 1 is not the "owner." Instead, it is the actual segment address of the item controlled by the block. The word at byte 3 is the size in paragraphs of the controlled item. Bytes 8 through 15 contain the filename, padded with blanks, of the file from which a driver was loaded.

Possible codes used in these subsegment control blocks are as follows:

Table 3-1: Codes used in subsegment control blocks.

Code	Directive	Meaning
D	DEVICE=	device driver
E		device driver appendage
I		IFS (Installable File System) driver
F	FILES=	control block storage area (for FILES>5)
X	FCBS=	control block storage area, if present
C	BUFFERS	EMS workspace area (if BUFFERS /X option used)
B	BUFFERS=	storage area
L	LASTDRIVE=	drive info table storage area
S	STACKS=	code and data area

Because these subsegment control blocks appear only in the DOS data segment and are meaningless elsewhere, they appear to be of limited use. Their purpose appears to be simplification of the MEM command introduced in DOS 4.0, although most of the information contained in them is duplicated elsewhere in each of the applicable DOS structures.

Actually, similar abbreviations are found in the buffer used internally by DOS for parsing CONFIG.SYS: 'X' represents FCBS=, for example, and 'D' represents DEVICE=. The abbreviations are not identical, however. For example, the CONFIG.SYS buffer uses 'K' to represent STACKS=, since 'S' apparently is needed for the SHELL= statement. (For more information on the CONFIG.SYS

buffer, see Michael J. Mefford, "Choose CONFIG.SYS Options at Boot," *PC Magazine*, 29 November, 1988, pp. 323-344: a fascinating article explaining a brilliant DOS utility.)

How to Find the Start of the MCB Chain

The key to locating any MCB is in the undocumented DOS List of Lists, whose address is retrieved with INT 21h Function 52h. Although the List of Lists returned by this function differs significantly from one version of DOS to the next, the MCB pointer's location is one of the very few items that is the same in all DOS versions to date. It's always located two bytes *in front* of the pointer returned in ES:BX, that is, at ES:[BX-2].

The value located there, however, is actually not a pointer to the first MCB, but its segment number. To use it as a pointer, you must provide an offset of 0000.

The following assembly language code fragment shows how to force the MCB pointer into ES:SI so that it may be used to retrieve the key byte, the owner word, and the size word:

```
        mov     ah, 52h          ; Get List of Lists
        int     21h
        mov     ax, es:[bx-2]    ; First MCB Segment Address
        mov     es, ax
        xor     si, si           ; force offset to be zero
```

or, to be safe:

```
        xor     bx, bx
        mov     es, bx          ; set ES:BX to 0:0
        mov     ah, 52h
        int     21h
        mov     cx, es
        or      cx, bx          ; is ES:BX still 0:0?
        jz      fail            ; then Function 52h not supported
        mov     ax, es:[bx-2]    ; First MCB Segment Address
        mov     es, ax
        xor     si, si           ; force offset to be zero
fail:
```

The next code sequence retrieves the key byte, the owner word, and the size word, respectively; it assumes that ES:SI is unchanged from the preceding example:

```
        mov     al, es:[si]       ; gets key byte, 'M' or 'Z'
        mov     bx, es:[si+1]     ; gets owner word or 0000
        mov     cx, es:[si+3]     ; gets size in paragraphs
```

For most applications, the following code fragment may be more useful; it can be used in Microsoft C 5.0 and higher, QuickC 2.0 and higher, and Borland Turbo C 2.0 and higher:

```
typedef struct {
    unsigned char type;          /* 'M'=in chain; 'Z'=at end */
    unsigned owner;              /* PSP of owner */
    unsigned size;               /* in 16-byte paragraphs */
    unsigned char unused[3];
    unsigned char dos4[8];
    } MCB;

#ifndef MK_FP
#define MK_FP(seg, ofs) \
     ((void far *) (((unsigned long)(seg)<<16) | (ofs)))
#endif

MCB far *Get_First_MCB( void )  /* locates first MCB   */
{ union REGS reg;
  struct SREGS seg;
  unsigned *tmpp;

  segread( &seg );              /* set up seg regs     */
  reg.h.ah = 0x52;              /* get List of Lists   */
  intdosx( &reg, &reg, &seg );
  tmpp = (unsigned far *) MK_FP( seg.es, reg.x.bx - 2 );
  return (MCB far *) MK_FP( *tmpp, 0 );
}
```

Get_First_MCB() is functionally identical to the first assembly-language fragment. It uses the segread() library function to avoid modifying the values of DS and SS while getting the List of Lists pointer in ES, and uses the MK_FP() macro to create the returned pointer value, rather than stuffing the appropriate quantities into ES and SI. As explained in chapter 2, you can also use in-line assembler or register pseudo-variables, if your compiler supports these options.

How to Trace the MCB Chain

Let's look at how to build a program, MEM, which you may already have on your machine: versions of this popular utility include PMAP (Chris Dunford), MAPMEM (TurboPower Software), TDMEM (Borland Turbo Debugger 2.0), and the commands MEM /PROGRAM and MEM /DEBUG in DOS 4.0 and higher. A program of this type walks through the MS-DOS MCBs. Because some MCBs control PSPs, the program can be used to trace through all PSPs, showing which programs are resident in memory. For example, here is sample output from MAPMEM on a Compaq 386 running Quarterdeck QEMM, making extensive use of LOADHI (CHKDSK typically reports 704,880 bytes free on this system!):

```
Allocated Memory Map - by TurboPower Software - Version 2.9

PSP   blks bytes owner    command line        hooked vectors
----- ---- ----- -------- ------------------- ------------------
0008   1    3904 config
0AE9   2    3776 command                      22 2E
0BEA   2   18432 TSREXAMP                     09 28 2F F1 FA
106B   2  686416 free
```

When we refer to MCBs controlling PSPs, we merely mean that the block of memory controlled by an MCB happens to be a program. (To be precise, it is not a program, but a *process*: a program that has been loaded into memory.) All DOS processes begin with a 256-byte (16-paragraph) PSP. The MCB controls the PSP only in the sense that the MCB is the arena header for the memory used by the PSP and by the process itself. For example, in the display above, the PSP at 0AE9 is "controlled" by an MCB at 0AE8. In turn, the "owner" field of the MCB at 0AE8 would be 0AE9.

Our MEM program will display the segment number of each MCB, the Program Segment Prefix (PSP) of its owner, and the size of the MCB (in hex paragraphs and decimal bytes). For MCBs that hold actual PSPs, MEM also displays the segment for the corresponding environment, the ASCII filename of the owner (which in DOS 3.0 and higher is kept in a program's environment), and any interrupt vectors that point into the block of memory. Here is what MEM's output looks like:

```
C:\UNDOC\KYLE>mem
Seg     Owner   Size            Env
09F3    0008    00F4 (  3904)           config [15 4B 67 ]
```

```
0AE8    0AE9    00D3 (  3376)   0BC1    c:\dos33\command.com [22 23 24 2E ]
0BBC    0000    0003 (    48)           free
0BC0    0AE9    0019 (   400)
0BDA    0BEA    000E (   224)
0BE9    0BEA    0472 ( 18208)   0BDB    C:\UNDOC\KYLE\..\rmichels\TSREXAMP.EXE
[09 28 2F F1 FA ]
105C    106B    000D (   208)
106A    106B    1218 ( 74112)   105D    C:\UNDOC\KYLE\MEM.EXE [00 ]
2283    0000    957C (612288)           free [30 F4 F5 F8 ]
```

INTs 15h, 4Bh, and 67h point into the "config" block because QEMM is loaded with a DEVICE=QEMM.SYS statement in CONFIG.SYS. INT 67h is used for the Expanded Memory Specification (EMS), INT 4Bh is used for "DMA Services" implemented by QEMM and other 386 memory managers (it is also present in newer PS/2 BIOSes), and INT 15h is taken over to control access to extended memory.

Before running MEM, we also ran the program TSREXAMP from Ray Michels's chapter. We can see that Ray's program takes about 18,000 bytes of memory, and that it hooks INTs 09h, 28h, and 2Fh. (But why are F1h and FAh pointing in there?!) In this example, we ran an early version of Ray's TSR that didn't free its environment: that's why the program name is still accessible. We also see that the filename stored in a program's environment is not reduced to its canonical form. The filename could be reduced to its canonical form, (C:\UNDOC\RMICHELS\TSREXAMP.EXE), using undocumented INT 21h Function 60h, which is discussed in chapter 4.

One limitation of many MCB walkers is that they assume the presence of only one MCB chain. In fact, programs such as 386MAX and QEMM allow memory resident programs to be loaded "high" by creating secondary MCB chains in high DOS memory. You can view these secondary MCB chains only by running PMAP or MAPMEM inside the LOADHI utility (for example, C:\QEMM>LOADHI PMAP). In the version of MEM developed here, we will allow the user to specify a segment on the DOS command line that can be used as the address of a possible secondary MCB chain. For example:

```
C:\UNDOC>mem BC00
Seg     Owner   Size            Env
BC00    BE80    0004 (    64)
BC05    BC06    0279 ( 10128)
BE7F    BE80    0237 (  9072)   BC01    C:\DOS\MOUSE.COM [0B 10 33 ]
```

```
COB7    COBD    0004 (    64)
COBC    COBD    055E ( 21984)   COB8    C:\CED\CED.COM [1B 21 61 ]
C61B    0000    21E4 (138816)           free
```

This shows there is a secondary MCB chain at BC00, both MOUSE.COM and CED.COM have been loaded "high," and there is still 138,000 bytes free of high DOS memory.

It is useful to write MEM in two stages: first, just print out raw information about DOS memory control blocks. Then, after that simple program is working, write an improved version that displays the ASCII filenames of the owners of the MCBs (which gives us a display of all programs resident in memory including, of course, the MEM program itself). Here is our first version of the MEM utility:

```
/*
MEM.C -- walks DOS MCB chain(s): simple version
Andrew Schulman and Jim Kyle, July 1990
*/

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef unsigned long ULONG;
typedef void far *FP;

#ifndef MK_FP
#define MK_FP(seg,ofs)  ((FP)(((ULONG)(seg) << 16) | (ofs)))
#endif

#ifdef __TURBOC__
#define ASM asm
#else
#define ASM _asm
#endif

#pragma pack(1)

typedef struct {
    BYTE type;          /* 'M'=in chain; 'Z'=at end */
    WORD owner;         /* PSP of the owner */

    WORD size;          /* in 16-byte paragraphs */
```

```
    BYTE unused[3];
    BYTE dos4[8];
    } MCB;

void fail(char *s) { puts(s); exit(1); }

MCB far *get_mcb(void)
{
    ASM mov ah, 52h
    ASM int 21h
    ASM mov dx, es:[bx-2]
    ASM xor ax, ax
    /* in both Microsoft C and Turbo C, far* returned in DX:AX */
}

void display(MCB far *mcb)
{
    char buf[80];
    sprintf(buf, "%04X    %04X    %04X (%6lu)",
        FP_SEG(mcb), mcb->owner, mcb->size, (long) mcb->size << 4);
    if (! mcb->owner)
        strcat(buf, "   free");
    puts(buf);
}

void walk(MCB far *mcb)
{
    printf("Seg     Owner   Size\n");
    for (;;)
        switch (mcb->type)
        {
            case 'M' : /* Mark : belongs to MCB chain */
                display(mcb);
                mcb = MK_FP(FP_SEG(mcb) + mcb->size + 1, 0);
                break;
            case 'Z' : /* Zbikowski : end of MCB chain */
                display(mcb);
                return;
            default :
                fail("error in MCB chain");
        }
}

main(int argc, char *argv[])
{
    if (argc < 2)
        walk(get_mcb());            /* walk "normal" MCB chain */
```

```
    else
    {
        unsigned seg;
        sscanf(argv[1], "%04X", &seg);
        walk(MK_FP(seg, 0));        /* walk arbitrary MCB chain */
    }
    return 0;
}
```

This code simply displays the raw MCB chain. The function get_mcb(), written with in-line assembler, returns a far pointer to the first MCB. Even though we're calling undocumented DOS Function 52h here, we don't bother to check DOS version numbers because the segment of the first MCB is *always* located at offset -2 in the List of Lists. It's even supported in the DOS compatibility box of OS/2 1.1 (DOS version 10.10). The start of the MCB chain is passed to the function walk(), which goes into an infinite loop, displaying an MCB and moving to the next MCB, until the end of the chain (or an error) is found. The MCB is displayed using the function display(). The output of this program looks like this:

```
Seg     Owner   Size
09F3    0008    03E1 ( 15888)
0DD5    0DD6    00D3 (  3376)
0EA9    0000    0003 (    48)    free
0EAD    0DD6    0040 (  1024)
0EEE    C0D6    0004 (    64)
0EF3    0F02    000D (   208)
0F01    0F02    1204 ( 73792)
2106    0000    7EF9 (520080)    free
```

MCB Consistency Checks

Actually, this code is useful by itself. It can be linked into other programs (with the exception of main()) and used to track their DOS memory allocation. This is particularly useful when you are trying to debug a program that trashes the MCB chain. By modifying the walk() function, you can check the MCB chain for consistency before DOS does. The chain is inconsistent if mcb->type is equal to anything other than 'M' or 'Z':

```
mcb_chk(MCB far *mcb)
{
    for (;;)
        if (mcb->type == 'M')
```

```
            mcb = MK_FP(FP_SEG(mcb) + mcb->size + 1, 0);
        else
            return (mcb->type == 'Z');
}
```

With mcb_chk(), a program can periodically check the MCB chain with a call such as the following:

```
if (! mcb_chk(get_mcb()))
{
    /* maybe do stack backtrace here, or dump registers */
    puts("Error in MCB chain - prepare for halt...");
    getchar();
}
```

Of course, if mcb_chk() *does* return false, then the next time any memory allocation is performed, the system will halt with a message such as:

```
Memory allocation error
Cannot load COMMAND, system halted
```

DOS merely performs the same consistency check as mcb_chk(), except that, if it does find anything other than 'M' or 'Z,' it has no choice but to halt the system. There seems to be no way that the MCB chain could be reliably repaired. In multitasking 80386 control programs such as DESQview or Windows 3.0, though, trashing the MCB chain in a "DOS box" (virtual machine) is far less catastrophic: you just throw the virtual machine away and get a new one.

Our minimal MCB walker has one other use. We can use it to reveal a bug in DOS itself. In the entry for INT 21h Function 4Ah (Resize Memory Block), Appendix A notes that "if there is insufficient memory to expand the block as much as requested, the block will be made as large as possible." Don't believe it? Just substitute the following main() for the one provided earlier:

```
main(void)
{
    unsigned segm;
    ASM mov ah, 48h     /* Allocate Memory Block */
    ASM mov bx, 64h     /* get 100 paragraphs */
    ASM int 21h
    ASM jc done
    /* ax now holds initial segment of allocated block */
```

```
    ASM mov segm, ax
    printf("before: "); display(MK_FP(segm - 1, 0));

    ASM mov ax, segm
    ASM mov es, ax       /* now resize the block */
    ASM mov ah, 4Ah      /* Resize Memory Block */
    ASM mov bx, 0FFFFh   /* impossible (at least in real mode!) */
    ASM int 21h
    ASM jnc done         /* something seriously wrong if _didn't_ fail!
*/
    printf("after:  "); display(MK_FP(segm - 1, 0));
done:
    return 0;
}
```

The resulting display shows that all remaining memory has in fact been given to the block, even though the call failed:

```
before: 1D4C    0BEA    0064 (  1600)
after:  1D4C    0BEA    9AB3 (633648)
```

The enormous number of bytes allocated to MCB 1D4C in the second line shows that, even though Function 4Ah returned with the carry flag set, indicating an error, the block was still made as large as possible. (It's particularly large here because this test was run on a system with Quarterdeck QEMM.) This is definitely a *bug* in DOS, *not* an undocumented feature on which you should depend! As it stands, reallocations that fail but that nonetheless snarf memory can cause mysterious program behavior.

This example also shows that the display() function can be useful all by itself: just pass it an MCB and it displays some useful information. Given the segment address of a block of memory, though, remember that the MCB is located at the *preceding* paragraph. If a PSP, for instance, is 1234h, its MCB is 1233h. This is why segm - 1, rather than segm, is used above in the call to display().

A More Detailed MEM Program

To produce a more complete display, we need only change the display() function, and add supporting functions and macros:

```
void display(MCB far *mcb)
{
    static void far *vect_2e = (void far *) 0;
```

```
    unsigned env_seg;

    printf("%04X    %04X    %04X (%6lu)   ",
        FP_SEG(mcb), mcb->owner, mcb->size, (long) mcb->size << 4);

    if (IS_PSP(mcb))
    {
    void far *e = env(mcb);  /* MSC wants lvalue */
        if (env_seg = FP_SEG(e)) printf("%04X    ", env_seg);
        else                             printf("        ");

        display_progname(mcb);
    }

    if (! vect_2e)
        vect_2e = GETVECT(0x2e);   /* do just once */
    if (! mcb->owner)
        printf("free ");
    /* 0008 is not really a PSP; belongs to CONFIG.SYS */
    else if (mcb->owner == 8)
        printf("config ");
    /* INT 2Eh belongs to master COMMAND.COM (or other shell) */
    else if (belongs(vect_2e, FP_SEG(mcb), mcb->size))
        printf("%s ", getenv("COMSPEC"));

    /* presence of command line is independent of program name */
    if (IS_PSP(mcb))
        display_cmdline(mcb);
    display_vectors(mcb);
    printf("\n");
}
```

The new display() calls env() to find out if the MCB contains the PSP of its owner and therefore has an associated environment block. Some of the relationships between MCB, PSP, and environment can get a little confusing, so we also use a few simple macros:

```
#define MCB_FM_SEG(seg)      ((seg) - 1)
#define IS_PSP(mcb)          (FP_SEG(mcb) + 1 == (mcb)->owner)
#define ENV_FM_PSP(psp_seg) (*((WORD far *) MK_FP(psp_seg, 0x2c)))

char far *env(MCB far *mcb)
{
    char far *e;
    unsigned env_mcb;
```

```
    unsigned env_owner;

    /*
        if the MCB owner is one more than the MCB segment then
            psp := MCB owner
            env_seg := make_far_pointer(psp, 2Ch)
            e := make_far_pointer(env_seg, 0)
        else
            return NULL
    */
    if (IS_PSP(mcb))
        e = MK_FP(ENV_FM_PSP(mcb->owner), 0);
    else
        return (char far *) 0;

    /*
       Does this environment really belong to this PSP? An
       environment is just another memory block, so its MCB is
       located in the preceding paragraph. Make sure the env
       MCB's owner is equal to the PSP whose environment this
       supposedly is! Thanks to Rob Adams of Phar Lap Software
       for pointing out the need for this check; this is a
       good example of the sort of consistency check one must
       do when working with undocumented DOS.
    */
    env_mcb = MCB_FM_SEG(FP_SEG(e));
    env_owner = ((MCB far *) MK_FP(env_mcb, 0))->owner;
    return (env_owner == mcb->owner) ? e : (char far *) 0;
}
```

The env() function uses the IS_PSP() macro, which tells whether an MCB corresponds to a PSP by verifying that the next paragraph in memory is the MCB's owner. env() further makes sure we don't pick up a stray environment for a program that has *freed* its environment: usually such programs don't bother to zero out the environment segment number located at offset 2Ch in the PSP.

Next, display() calls display_progname(), which in turn calls progname_fm_psp(), a useful utility function that, given a PSP, tries to return a far pointer to the name of the corresponding program:

```
char far *progname_fm_psp(unsigned psp)
{
    char far *e;
    unsigned len;
    /* is there an environment? */
```

```
    if (! (e = env(MK_FP(MCB_FM_SEG(psp), 0))))
        return (char far *) 0;

    /* program name only available in DOS 3+ */
    if (_osmajor >= 3)
    {
        /* skip past environment variables */
        do e += (len = fstrlen(e)) + 1;
        while (len);

        /*
            e now points to WORD containing number of strings following
            environment; check for reasonable value: signed because
            could be FFFFh; will normally be 1
        */
        if ((*((signed far *) e) >= 1) && (*((signed far *) e) < 10))
        {
            e += sizeof(signed);
            if (isalpha(*e))
                return e; /* could make canonical with INT 21h AH=60h */
        }
    }

    return (char far *) 0;
}

void display_progname(MCB far *mcb)
{
    char far *s;
    if (IS_PSP(mcb))
        if (s = progname_fm_psp((FP_SEG(mcb) + 1)))
            printf("%Fs ", s);
}
```

If an MCB corresponds to a PSP [IS_PSP() is TRUE], display_progname() calls progname_fm_psp(), which first verifies if there is an environment. There is possibly a little *too* much verification and double-checking in this program, but any program that traffics in undocumented DOS should definitely be more paranoid than programs that rely only on documented interfaces. In DOS 3+, prog name_fm_psp() walks past all variables in the environment to find the ASCIIZ pathname of the program owning the environment (see the description of the DOS environment block in the appendix entry for INT 21h Function 26h).

The new version of display() next performs a number of tests to see if the block is free, if it is the very first block allocated by DOS (at CONFIG.SYS time),

or if it belongs to the master copy of COMMAND.COM. As explained in the chapter on command interpreters, undocumented INT 2Eh points into the master copy of COMMAND.COM. The simple function belongs() is used to find if an interrupt vector points into the block controlled by a given MCB:

```
typedef enum { FALSE, TRUE } BOOL;

BOOL belongs(void far *vec, unsigned start, unsigned size)
{
    unsigned seg = FP_SEG(vec) + (FP_OFF(vec) >> 4); /* normalize */
    return (seg >= start) && (seg <= (start + size));
}
```

Next, display() calls display_cmdline():

```
void display_cmdline(MCB far *mcb)
{
    /*
        psp := MCB owner
        cmdline_len := psp[80h]
        cmdline := psp[81h]
        print cmdline (display width := cmdline_len)
    */
    int len = *((BYTE far *) MK_FP(mcb->owner, 0x80));
    char far *cmdline = MK_FP(mcb->owner, 0x81);
    printf("%.*Fs ", len, cmdline);
}
```

Note that display_cmdline() uses the C printf() mask "%.*Fs" to display a far string, using the maximum length given by the variable len (whose value may be zero). Sometimes garbage is printed by MEM, or by any similar program, because the disk transfer area (DTA) located inside the PSP overlays the beginning of the command line.

Finally, display() calls display_vectors() to show any interrupts hooked by the program whose PSP is contained in this MCB. The function finds these hooked interrupt simply by seeing if CS:IP for the interrupt handler falls within this MCB:

```
#ifdef __TURBOC__
#define GETVECT(x)      getvect(x)
#else
```

```
#define GETVECT(x)      _dos_getvect(x)
#endif

void display_vectors(MCB far *mcb)
{
    static void far **vec = (void far **) 0;
    WORD vec_seg;
    int i;
    int did_one=0;

    if (! vec)
    {
        if (! (vec = calloc(256, sizeof(void far *))))
            fail("insufficient memory");
        for (i=0; i<256; i++)
            vec[i] = GETVECT(i);
    }

    for (i=0; i<256; i++)
        if (vec[i] && belongs(vec[i], FP_SEG(mcb), mcb->size))
        {
            if (! did_one) { did_one++; printf("["); }
            printf("%02X ", i);
            vec[i] = 0;
        }
    if (did_one) printf("]");
}
```

In DOS 4.0 and higher, some memory-resident software can be loaded using the INSTALL= statement in CONFIG.SYS. Such programs can show up in the MEM display as MCBs that aren't associated with any program but that may have hooked interrupt vectors. Note that MEM calls display_vectors() for all MCBs, even when there seems to be no associated program. For example, in DOS 4.0 and higher, if CONFIG.SYS contains the statement INSTALL= C:\CED\CED.COM to load Chris Dunford's CED command-line editor, then MEM will display something like the following:

```
0E81    0E82    065F ( 26096)   [1B 21 61 ]
```

Another benefit of calling display_vectors() for all MCBs is that occasionally we find "orphaned" interrupt vectors that point into free memory:

```
2A2A    0000    75D5 (482640)           free    [30 F4 F5 F8 ]
```

INT 30h is a far jump instruction, not an interrupt vector, but INTs F4h, F5h, and F8h are real interrupt vectors: let's hope no program invokes them while they're pointing into free memory!

Finally, the following boring little function is used so that we can easily get the length of far strings, even from a small-model program:

```
unsigned fstrlen(char far *s)
{
#if defined (_MSC_VER) && (_MSC_VER >= 600)
    return _fstrlen(s);
#else
    unsigned len=0;
    while (*s++)
       len++;
       return len;
       #endif
}
```

We now have a fairly complete implementation of the MEM program.

Allocation Precautions

Each time DOS INT 21h Function 4Bh loads a program for execution, it allocates memory for it as well. For a COM-format file, the loader requests all available RAM. For an EXE-format file, the amount of RAM is specified in the file's relocation header. If this amount is not explicitly defined at link time, however, the EXE takes all available space.

Because most programs therefore hog all RAM each time they are loaded, whether they need it or not, it's up to you as a programmer to be sure that your programs trim themselves back to no more than they need, if they are going to be spawning other processes. Failure to do so will result in "out of memory" errors no matter how much RAM your system contains.

Each time a program terminates normally and returns control to its parent process, all RAM allocated to that program is once again made available for allocation. If the program terminates via one of the TSR functions, only part (or possibly none) of its memory is released to be used again.

The upshot is that programs get all available space while they are executing, and can turn it back when they finish. Memory allocation for your programs can thus be handled automatically (and invisibly) by DOS itself. Unfortunately, getting one large block of memory at start-up, and having it deallocated for you at

termination, is often inconvenient because it means your program can't spawn other processes. Modern C compilers usually include, in their start-up code, the necessary calls to cut their RAM usage back to just what they require, or 64K, whichever is larger, but at least one popular high-level language (Turbo Pascal) does not do this automatically. Instead, TP gives you a compiler option ("$M") that lets you specify how much memory to use.

Unfortunately, this option (like the DOS loader) defaults to "all available space," so an error condition will result from attempts to EXEC or SPAWN a child process from TP without using the $M option to make RAM available for the child process. Because of this, the Turbo Pascal Exec() procedure gained a reputation for being broken; actually, it just was not adequately documented.

If you happen to be writing in assembly language, it's entirely your responsibility to manage your memory allocations properly.

In most high-level language programming, you won't use the three DOS RAM allocation functions directly, but if you use the C library functions malloc() and free(), or the Turbo Pascal functions new() and dispose(), you'll be using them indirectly.

The strategy behind malloc() and free() is to obtain large blocks of RAM using the DOS functions and then dole it out to the program in much smaller portions, as requested. This is a heritage from UNIX, where the allocation of system RAM was a time-consuming process. Under MS-DOS, the reverse is true, and a number of "improved performance" packages that replace the standard library versions of malloc() and free() with more-direct calls to the DOS functions have appeared recently.

On the other hand, each block of memory allocated from DOS requires the additional 16-byte MCB (arena), and all DOS allocations are consequently paragraph-based, so if you want to allocate 4 bytes from DOS, for instance, you have to ask INT 21h Function 48h for one paragraph (16 bytes). In order to satisfy this request, DOS then actually needs *two* paragraphs: the one you asked for, plus one for the MCB to "control" the paragraph. The point is that the smallest possible direct DOS memory allocation actually uses 32 bytes.

The Turbo Pascal new() and dispose() functions are more direct than the C malloc() and free() functions: they depend on your program having already taken all available RAM from DOS, and then simply maintain a pointer to the lowest unused byte in the "heap" (the unused data area above your program's minimum requirement). A call to new(), with the number of bytes needed supplied as its

argument, returns the current value of that pointer, and increments the pointer past the number requested so that the next call to new() will get still-available space. A call to dispose(), which takes a pointer as its argument, simply sets that pointer into the FreeHeap variable that new() uses, thus making all RAM above that address available for reallocation.

RAM Allocation Strategies

When a program requests x paragraphs of memory and the memory manager has more than one block free, it's possible to satisfy that request in several different ways. These different ways of allocating memory are known as "allocation strategies," and DOS provides a function (INT 21h Function 58h) to select from three different strategies.

This function isn't always documented, however. For example, it is described in Microsoft's *MS-DOS Encyclopedia*, but not in *IBM's Technical Reference* for DOS 3.3. Thus, like several other functions, it's neither officially documented across the board nor truly undocumented. We'll refer to it as "semi-documented." At any rate, the function permits you to select a "first fit," "best fit," or "last fit" strategy for the memory manager to follow.

First-fit Strategy The first-fit strategy is optimized for speed, with the possible result of excessively fragmenting memory. In a predominantly single-process system (which is the normal condition of DOS), such fragmentation is unlikely, though, so this is the default action unless you explicitly change things.

Even if you do change strategies, DOS will change back to first-fit whenever it loads a program, although it follows your selected strategy for all other loading actions.

When using the first-fit strategy, the memory manager begins looking for free RAM at the start of the MCB chain and makes the allocation from the first block it finds that is large enough to satisfy the request. If the block is larger than requested, only enough is taken off the front to fill the request, and a new, still free, block is created for the remainder.

In normal everyday DOS operation, there's usually only one such block in the system when a program is loaded. Because the loader often asks for "all available" RAM, no new block is created. Under these conditions, there's no difference between first-fit and best-fit strategies.

If, however, the available RAM has become highly fragmented, and at the same time the block being allocated is small enough to fit in the first free block encountered, the first-fit strategy will use that first block.

Best-fit Strategy The best-fit strategy must continue all the way to the end of the MCB chain. This strategy is optimized for tightest use of RAM space, without regard to operating speed; such an approach is sometimes essential. Unlike the first-fit strategy, which accepts the first usable space, the best-fit strategy requires that all available RAM be examined and then allocates the request from the *smallest* block that will do the job, regardless of whether it is the first one encountered. As with the first-fit strategy, the block is allocated from the front, and any left-over space is put into a new, still free, block.

This approach guarantees that multiple allocations of small blocks will not fragment RAM unnecessarily. So long as blocks are released at approximately the same rate as they are allocated, the best-fit strategy will continue using the same small blocks over and over, leaving the larger blocks free to accommodate requests that require them.

As pointed out in the previous section, in normal operation with only one or two blocks of RAM free, there's little difference in action between first-fit and best-fit. If, however, you are programming an application that does its own RAM management and that makes short-term use of large numbers of small blocks of RAM, you'll want to keep this strategy in mind. It could keep you from running out of RAM unexpectedly just because none of the remaining free blocks is large enough to fill your latest request!

Having said this, it is important not to oversell best-fit. In fact, as any textbook on operating systems will tell you, first-fit is almost always the correct strategy to use.

Last-fit Strategy Unlike either of the other strategies, the last-fit technique is designed specifically for allocations that you expect to hang around for a long time, such as TSRs or device drivers. Unfortunately, DOS doesn't let you use the last-fit technique to load programs.

When a block of RAM is allocated using the last-fit strategy, the highest possible block of memory that can satisfy the request is assigned. Normally this will be the highest part of the final block of free RAM. The idea is that memory allocated at the end of the MCB chain won't ever need to be searched, if you switch back to the default first-fit strategy for subsequent normal allocations.

Because the DOS loader won't honor this strategy for loading program material, the last-fit strategy is of limited usefulness. You can use it to force things to the top of the normal 640KB conventional-RAM area, though.

Selecting The Strategy

The following C language sample program (STRATST.C) illustrates the use of the semi-documented strategy function and verifies its operation. Like the previous sample programs in this chapter, it compiles with either Microsoft or Turbo C.

Note that in STRATST.C each different DOS action has been encapsulated into a unique function. This not only makes it easier for you to extract them into your own programs but also simplifies the logical flow of the illustration itself.

```
/*
STRATST.C - Jim Kyle - June 5, 1990
demonstrates RAM-management strategy function
*/

#include <dos.h>
#include <stdio.h>

union REGS reg;
struct SREGS seg;
unsigned bigblok, tinyblok;
char *codes[] = { "First-fit", " Best-fit", " Last-fit" };

unsigned int GetRam ( unsigned para )   /* paragraphs!  */
{ reg.x.bx = para;
  reg.x.ax = 0x4800;            /* get RAM from DOS     */
  intdos( &reg, &reg );
  printf("Got blk at %04X, size=%u\n", reg.x.ax, reg.x.bx );
  return reg.x.ax;
}

void RelRam ( unsigned segmt )  /* release RAM to DOS   */
{ segread( &seg );
  seg.es   = segmt;
  reg.x.ax = 0x4900;
  printf("\nReleased block at %04X", seg.es );
  intdosx( &reg, &reg, &seg );
}

void SetStrat ( char strat )
{ reg.x.ax = 0x5801;            /* set strategy code    */
```

```
  reg.h.bl = strat;
  intdos( &reg, &reg );
}

void GetStrat ( void )
{ reg.x.ax = 0x5800;              /* read strategy code   */
  intdos( &reg, &reg );
  printf("\nStrategy code: %u (%s): ",
         reg.x.ax, codes[reg.x.ax] );
}

void main ( void )
{ bigblok = GetRam ( 0x100 );     /* allocate big block   */
  GetRam ( 0x080 );               /* allocate a fence     */
  tinyblok = GetRam ( 0x080 );    /* allocate tiny block  */
  GetRam ( 0x080 );               /* allocate a fence     */

  RelRam ( bigblok );             /* now free two blocks  */
  RelRam ( tinyblok );            /* but leave fences     */

  SetStrat ( 0 );                 /* set first-fit        */
  GetStrat ();
  RelRam ( GetRam ( 0x80 ));      /* get, then release    */

  SetStrat ( 1 );                 /* set best-fit         */
  GetStrat ();
  RelRam ( GetRam ( 0x80 ));      /* get, then release    */

  SetStrat ( 2 );                 /* set last-fit         */
  GetStrat ();
  RelRam ( GetRam ( 0x80 ));      /* get, then release    */

  SetStrat ( 0 );                 /* set first-fit at end */
}
```

The program assigns four consecutive blocks of RAM, with the first of the four twice as large as each of the other three. The first and third blocks are then released to create artificial fragmentation so that you can see the effects of the different strategy selections. Aside from any small blocks of free space that might exist as leftovers from TSR installation, your system should at this point have three blocks of free RAM available for allocation: the 4,096-byte (0x100 paragraphs) block allocated as "bigblok," the 2,048-byte block allocated as "tinyblok," and "all the rest" between the second "fence" block and the top of memory.

With this preparation, STRATST.C sets the first-fit strategy via SetStrat(), verifies it by means of GetStrat(), and then allocates and immediately frees a block of 2,048 bytes (0x80 paragraphs). GetRam reports the address of the block obtained; releasing it restores the status quo for the next test.

Similarly, the best-fit and the last-fit strategies are tested. Before returning to DOS, the program restores first-fit as the strategy to be used (failure to do so in early test versions led to startling reports from other demonstrations—such as environments appearing in memory *after* their respective programs—although everything continued to work properly).

Here is the report generated by STRATST; the exact addresses will differ for your system, but the results should be similar:

```
Got blk at 89E9, size=256
Got blk at 8AEA, size=128
Got blk at 8B6B, size=128
Got blk at 8BEC, size=128
Released block at 89E9
Released block at 8B6B
Strategy code: 0 (First-fit): Got blk at 89E9, size=128
Released block at 89E9
Strategy code: 1 ( Best-fit): Got blk at 8B6B, size=128
Released block at 8B6B
Strategy code: 2 ( Last-fit): Got blk at 9F80, size=128
Released block at 9F80
```

In addition to printf() statements embedded directly in the program, we could also have viewed this program's memory allocation using the INTRSPY program, presented in chapter 8. That chapter includes an INTRSPY script for tracking calls to INT 21h Functions 48h, 49h, and 4Ah.

Process Management

The concept of a "process" as a separate executable program that has been loaded into memory but that may or may not be executing currently is central to the operation of MS-DOS. The whole basis of TSR programming is that a process may be retained "in residence" after once terminating, but TSRs are not the only processes that DOS manages. Every program loaded for execution, including the command interpreter itself, is a process.

The PSP: How It Identifies a Process

Even when we are discussing DOS memory management, there was no way to avoid mentioning the Program Segment Prefix (PSP). Now we can examine this crucial DOS data structure in more detail. The PSP, a 256-byte block located immediately preceding the actual process memory, is the key to process management in MS-DOS. The PSP contains the DOS state (file handles, etc.) for its process; the segment address of the PSP itself provides a unique identifier by which the process can be located and managed.

History, Purpose, and Use The Program Segment Prefix came to MS-DOS by way of Seattle Computer Products's 86-DOS, which, for compatibility, took the concept from the Digital Research's 8080 CP/M operating system.

As MS-DOS developed through the years, however, the PSP has evolved into far more than its CP/M equivalent. It now embodies many of the concepts provided in other operating systems (such as UNIX and Multics) by the "stack frame" or the "process directory." By proper use of information kept in the PSP, a process can pass data to other processes that it spawns, or it can return information back to its parent process. At the same time, many fields of the PSP are vestigial, holdovers from the days of CP/M.

The primary purpose of the PSP is to contain the system information necessary to start, run, and finish a specific process. This includes, but is not limited to, the address of the routine to which control should transfer when the process terminates, the list of "handles" by which the process identifies its files and devices, the address of the environment space belonging to the process, the identity of the process' parent process, and last but far from least, any arguments passed directly to the process when it was invoked.

A secondary purpose is to provide methods of accessing DOS functions without INT 21h; this was much more important in earlier times than it is today. With CP/M, the interface to BDOS (Basic Disk Operating System, the ancestor of the INT 21h functions) was by way of a subroutine call to location 0005h. Consequently, to provide the same functionality, offset 0005h in the PSP of every process contains a rather cryptically coded far jump to the dispatcher area of MS-DOS itself.

Similarly, many UNIX systems provided similar capabilities through a far call in the user's stack frame area, so with the introduction of UNIX-like capabilities in MS-DOS 2.0, a special far call to INT 21h was added to the PSP at offset 0050h.

Neither of these capabilities is widely used; most programs today simply use INT 21h or, if the program is coded in a high-level language, its equivalent.

Unique Process Identifier MS-DOS can have only one "current process," because it uses the associated PSP as a scratch-pad area for much of its file management activity. Yet MS-DOS *can* be used for multitasking between multiple processes. A key to multitasking in an operating system with only one "current process" is simply to *change* this current process.

Throughout much of the MS-DOS documentation, you'll find references to an entity called the "process identifier," often abbreviated to "process ID" or even "PID." This is a 16-bit value that uniquely identifies each process currently resident in the system, regardless of whether it is active. However the documentation never explains precisely what the PID is.

This mysterious "process identifier" is nothing more than the segment address of the PSP associated with that process. DOS provides two undocumented functions, and one documented one, to store or retrieve the PID of the "current process," thus activating one or another set of data stored in different PSPs. The current process is set using undocumented INT 21h Function 50h, and the current process can be queried either with undocumented INT 21h Function 51h or, in DOS 3.x and higher, with the equivalent documented Function 62h.

It is important to understand that the two Get PSP functions do not necessarily retrieve the PSP of the program that calls them. There appears to be a great deal of confusion on this point. For example, even Duncan's *Advanced MS-DOS Programming* states that Function 62h "allows a program to conveniently recover the PSP address at any point during its execution, without having to save it at program entry."

In fact, the two Get PSP functions always return the value that was last established with Set PSP. This corresponds to the "current process" in DOS, not necessarily to the PSP of the calling program. If Get PSP is called from a TSR that has been activated by an interrupt, Get PSP returns the PSP of the foreground process, *not* the TSR's PSP. That is what makes the Get/Set PSP functions important: they are the basis for the ability to switch between multiple tasks in MS-DOS. It is often said that DOS is single-tasking, but this merely means that only one process "owns" DOS *at any given time.*

Whenever the current process is switched, whether by your own multitasking code or by a TSR popping up for action, it's essential that the current PID also be switched if any I/O activity is to occur. Otherwise, the files or devices

"owned" by the old foreground process will be affected, rather than your own files.

The three get/set PSP functions are described in further detail in Ray Michels's chapter on TSRs and DOS multitasking.

Undocumented Areas of the PSP Less than 1/3 of the 256-byte area in the PSP has been documented officially; this section supplies information about the remaining parts. Not all of them, however, have ever been put to use.

This description is organized in the form of an assembly-language data segment, though without the SEGMENT directives:

```
FINI:   INT     20H             ;0000 CP/M-like exit point
NXTGRAF DW      0A000h          ;0002 first unused segment
        DB      0               ;0004 filler to align next
CPMCAL: CALLF   INT21DSP        ;0005 CP/M-like service call
ISV22   DD      0               ;000A documented ISR vectors
ISV23   DD      0               ;000E " (saved at start)
ISV24   DD      0               ;0012 "
PARENT  DW      PARENT_ID       ;0016 PSP of parent
HANDLES DB      1,1,1,0,2       ;0018 indices into SFT
        DB      15 DUP(255)     ;     maintained by DOS
ENVPTR  DW      ENVIRON         ;002C environment segment
SAVSTK  DD      0               ;002E saved SS:SP at INT21
NHDLS   DW      20              ;0032 nbr of handles avail
HTBLPTR DD      HANDLES         ;0034 ptr to handle table
        DD      -1              ;0038 SHARE's previous PSP
RSVD1   DB      14 DUP(0)       ;003C never used
DISP:   INT     21H             ;0050 Unix-like dispatcher
        RETF
RSVD2   DB      9 DUP(0)        ;     never used
FCB1    DB      0,'           ' ;005C documented FCB areas
        DB      0,0,0,0
FCB2    DB      0,'           ' ;006C
        DB      0,0,0,0
TAILC   DB      5               ;0080 "command tail" count
TAIL    DB      ' args'         ;0081 start actual data here
        DB      0DH
```

DOS Termination Address

Although the three interrupt service vectors saved in the PSP are documented, their usage is not, and one of them provides a way to hook into a process at termination time no matter what causes the process to terminate, thus providing

DOS "Exit List" capability. The magic vector is ISV22, the INT 22h vector, documented as the "termination address." What is not documented is the fact that the address in the PSP, rather than the one in the interrupt service region, is the one used when the process terminates!

To hook this vector and cause your own code to be executed when the process terminates, before control returns to the calling program, just use the following routines, with your own code inserted as noted. Execute "SetHook" during your program's initialization, while ES still points to the PSP (as it will upon entry to the program for an EXE file, or at any time before you change it for a COM file); "DoHook" will be called automatically at termination time:

```
SetHook PROC
        MOV     AX,[ES:000Ah]   ; save old offset
        MOV     word ptr CS:OldVec,AX
        MOV     AX,[ES:000Ch]   ; save old segment
        MOV     word ptr CS:OldVec+2,AX
        MOV     AX,offset DoHook
        MOV     [ES:000Ah],AX   ; set in new vector
        MOV     AX,CS
        MOV     [ES:000Ch],AX
SetHook ENDP

OldVec  DD      0               ; place for old vector

DoHook  PROC    FAR
; whatever you need to do is coded here...
        JMP     [CS:OldVec]     ; then chain to original
DoHook  ENDP
```

The "termination address" stored at ISV22 in the PSP is just the return address to the Exec function call (INT 21h Function 4B00h) that the parent used to invoke this process. Obviously, then, when DOS transfers control to this address, it is ready to return to the parent. DoHook is grabbing control instead, so when DoHook is executed, all memory allocated to the terminating process has already been released—including the memory containing the DoHook code. All files have been closed and the current PSP has been set to that of the parent. In DOS 3 and later, the registers have been restored to the values they had when the parent performed the Exec. Basically, all that DoHook should do is de-install any special handlers that the program had installed, so that they will not be left pointing to no-longer-valid addresses. *No other actions should be attempted* and by all means no

file access or other I/O should be tried since the context could easily vary depending on the parent programs. For more complex on-exit processing, you are better off using routines such as atexit() in C.

Other PSP Fields

The first fully undocumented area of the PSP is the word at offset 0016h, which contains the PID of this process' parent process. If this process is the current command interpreter, its own PID will appear here, even if it is really a spawned shell that can be terminated by the EXIT command. Were it not for this, you could trace back through these pointers from one PSP to the parent PSP and thus locate the master command interpreter. However, all you can do by tracing this is to locate the current shell, which may not be the master. (As noted earlier, INT 2Eh can be used to find the master copy of COMMAND.COM; more details are given in the chapter on Command Interpreters.)

Immediately following the PARENT pointer, at offset 0018h, is the 20-byte handle table. Each byte in this list represents an index into the System File Tables maintained by DOS. As shown in the example code, the first five of these are automatically set up by the loader routines to predefine handles for stdin, stdout, stderr, stdaux, and stdprn; note that the first three handles all reference the same System File Table (SFT; see the chapter on the DOS file system) entry for device CON. All unused handles have the value 0xFF.

The next undocumented area is the doubleword at offset 002Eh, which the DOS dispatch code uses to save SS and SP each time this process enters INT 21h. Saving the stack location in the PSP, rather than in DOS' own data area, makes multitasking possible by permitting DOS to switch current processes, resuming each process where it was last halted (that is, treating the processes as coroutines). However, MS-DOS itself has not yet taken advantage of this capability.

Right behind ENVPTR comes a 6-byte group added at version 3.1, which permits you to relocate the handle table and thus make more than 20 file handles available to your process. A documented DOS function (INT 21h Function 67h) exists to manipulate this area. An alternative to Function 67h appears in FHANDLE.C, in chapter 4.

The first two bytes of this region are the word NHDLS at offset 0032h, which defines the number of handles available to this process; attempting to open another file or device when this many handles are already in use will trigger a DOS error.

The following four bytes, HTBLPTR at offset 0034h, are a far pointer to the first byte of the handle table.

By default, NHDLS is set to 20, and HTBLPTR to PSP:0018h, thus describing the handle table in the PSP.

The doubleword at offset 0038h is always set to 0FFFF:FFFFh in DOS versions prior to 3.3. Later DOS versions set this to point to the parent PSP when SHARE is in use.

Spawning Child Processes

As is discussed further in chapter 6, every program run under MS-DOS can be thought of as a child process. Even the very first one loaded as part of the boot process (that is, the loader that is read in from the boot sector of the disk) is a "child" of the ROM Bootstrap routine! This section describes in detail the differences between a child process and the generic "process" concept.

A "child process" is simply a process spawned by some other process, which is called the "parent." Again, except for the bootstrap loader code that initially brings your system into action, every process in the system is a child of some other process.

The bootstrap loader spawns only one child: the command interpreter specified by the SHELL= line in CONFIG.SYS, or COMMAND.COM by default if no SHELL is specified. This process is what most users perceive to be DOS itself. Each time a program's name is typed on the command line, that program is spawned as a child of the command interpreter, for execution.

If the spawned program is, itself, a menu or other type of shell routine, it may in turn spawn children of its own, which execute and return control to their parent. Should control ever return to the bootstrap loader, the result is the error message "Bad or missing command interpreter" and a locked system requiring rebooting.

Locating Parent Processes

From time to time, a process needs to be able to trace its ancestry. This isn't always possible, because MS-DOS has a few quirks in some areas (for example, a shell program such as COMMAND.COM is always its own parent, and so the chain stops right there—see chapter 6 for more details). However, if the process is running as the child of anything *other* than a command interpreter such as COM-

MAND.COM, the job of locating its ancestors is straightforward though undocumented.

Locating Ancestors One undocumented field in the PSP, the PARENT word described previously, makes it possible for a program to trace its ancestry to the point of the closest command interpreter shell (any shell program modifies this field to show that it is its own parent).

Thus, a program that needs to trace its ancestry need only locate its own PSP, extract the PARENT process ID, then use that to access the parent's PSP. The process continues until the point at which PARENT points to the PSP that contains it; this will be the first command interpreter program encountered in the trace.

Use of this Capability A sample program in C that uses this capability to trace its ancestry follows:

```
/*
ROOTS.C (with apologies to Alex Haley)
Trace Your Ancestry!
Jim Kyle, 1990
*/

#include <stdio.h>
/* grr! different locations for _psp global variable! */
#ifdef __TURBOC__
#include <dos.h>
#else
#include <stdlib.h>
#endif

unsigned parent, self;

#define WORD(seg, ofs) \
     (*((unsigned far *) (((unsigned long)(seg)<<16) | (ofs))))

main ( void )
{ self = _psp;            /* start with own PSP value   */
  parent = WORD( self, 0x16 );
  do
    { printf("PID = %04X, PARENT = %04X\n", self, parent );
      self = parent;
    }
  while (( parent = WORD( self, 0x16 ) ) != self );
  return 0;
}
```

The program simply copies its own PID into the variable "self" and then uses it and the defined offset of PARENT in the PSP to retrieve the parent's PID in "parent."

From there, the program loops reporting the values of "self" and "parent" at each level and then redefining them both, until it reaches the level at which the two values match. This will be the command interpreter. At this point, the program returns. It's most instructive, by the way, to run this program from some environment, rather than from the command line, because that will guarantee at least one level of ancestry before the command interpreter is reached. For example, we can run ROOTS inside of DEBUG, inside another copy of DEBUG:

```
C:\UNDOC\KYLE>debug \bin\debug.exe roots.exe
-g
-g
PID = 932A, PARENT = 8F20
PID = 8F20, PARENT = 8B16
PID = 8B16, PARENT = 8A27
```

Here, 932A is ROOTS, and 8F20 and 8B16 are DEBUG. Naturally, we could use the code developed earlier in MEM, especially the function prog-name_fm_psp(), to find the ASCIIZ names of these ancestors.

Device Management

In addition to memory, the operating system must manage all devices connected to the CPU, such as the disk drives, the keyboard, and any displays. Although much of the detailed interface between any device and DOS is handled by the BIOS (Basic Input Output System) code (for example, BIOS INT 13h handles the disk), the actual management of devices remains the responsibility of DOS itself.

Why Device Drivers Exist

Older operating systems, and even MS-DOS 1.x, included all hardware-dependent code necessary to deal with input and output as an integral part of the system itself. This made it necessary to bring out version 1.1 of MS-DOS when IBM made available the 360KB double-sided floppy disk drive, and made it impossible to use any kind of hard disk conveniently on a DOS 1.x system. Improvements were obviously in order.

A major part of the upgrade provided by MS-DOS 2.0 was the "installable device driver" capability. This concept, borrowed from Bell Labs's UNIX, concentrates all hardware dependencies into small modules that can be installed and removed separately from the main operating system code itself.

An installable device driver is a code package that forms a self-contained unit capable of initializing itself, and through which all communication to and from a specific hardware device can be channeled. The format of the driver, and of its command interface, is rigidly specified by the MS-DOS documentation.

By separating hardware dependencies into such a module, only new drivers, rather than a complete operating-system upgrade, need be developed when a new hardware device becomes available; the new device is then immediately usable with any older system that can accept the driver.

Hardware Dependent Details

In general, three types of action tend to be highly device-specific and vary from one device to the next. These are the actions required to initialize the device and prepare it for use, those required to send data to it, and those required to receive data from it.

You might think that some devices need only two of these groups, because you don't usually send data to a keyboard or receive data from a printer. However, the keyboard does have to receive certain commands from the operating system to acknowledge that its output has been accepted, and similarly the system needs to read status conditions from the printer. These peripherals really are I/O devices, not just I or O devices.

Other details that are associated with specific hardware items rather than with generic logical functions include port addresses through which communication is achieved, the "handshake" protocol used to transfer data to and from the device, and the actual bit patterns transferred as commands and status.

All of these hardware-dependent details are concentrated within the single driver that serves each device. In order for DOS to use them, they are grouped into a small collection of logical functions as specified in the DOS documentation.

Logically Required Functions The DOS documentation specifies 17 logical functions, all of which must be recognized and responded to by every device driver, regardless of whether that function makes sense for the driver (as in the amusing case of "media check" for a CRT). These functions provide adequate flexibility to deal with virtually any I/O requirement you can imagine.

Normally function dispatching is implemented with a jump table. The function's code is used as the index into a table of offset addresses, and control transfers to the indexed address. If the specific function does not apply to this driver, the code reached normally returns an appropriate status code with no other action performed.

Congruence of Files and Devices One of the most useful results of the device driver concept is that MS-DOS can treat files and devices in exactly the same way. This means that you can write programs which deal simply with "streams" of data and not be at all concerned whether the streams come from (or go to) a device or a file. This is a major advantage when compared to older techniques that, to retrieve data from, for example, the keyboard, required totally different programming than that used to retrieve data from a file.

Unfortunately, not all the capability of the keyboard as an input device can be used through the drivers, nor can maximum display speed be obtained from the CRT. If you are programming a real-time video image display system, with hotkey control, you'll be forced to go direct to the video display controller with your output, and to use BIOS routines to read the keyboard without waiting until the operator presses ENTER.

Thus, not all programs which run under MS-DOS are able to take full advantage of the power offered by the driver concept. This is not a limitation inherent in the concept itself, but rather an artificial one imposed by the design of MS-DOS and failure to anticipate all future needs. Or maybe it's an inherent limitation in the concept of "device independence."

One interesting by-product of the files-devices congruence is that all your named devices can be accessed as files in any disk directory! This comes about because the DOS routines that *open* both devices and files always search for devices first, and if a device name is the same as the name of the file you are trying to open, the device will be opened instead. Because most of the procedures used to determine whether a given file exists depend on trying to open the file and then detecting the error if it cannot be opened, these routines will show that any device exists as a file in any directory you happen to test. Nevertheless, it will not show in the directory listing.

This can be used to test for the existence of a directory itself, because if you try to open a device by referring to it as a file in a nonexistent directory, the directory error will occur *before* the device access attempt. That error, in turn, indicates that the directory itself cannot be accessed, for if the directory can be accessed,

the device can also always be accessed. The following batch file uses this aspect of DOS devices:

```
@echo off
rem isdir.bat
if exist %1\nul goto exists
echo No such directory
goto done
:exists
echo Directory exists
:done

C:\UNDOC\KYLE>isdir \foobar
No such directory
C:\UNDOC\KYLE>isdir \undoc\kyle
Directory exists
```

Tracing the Driver Chain

In order to operate at all, MS-DOS must provide at least a minimal set of built-in device drivers. Yet to achieve the full advantages of expansion, it's necessary to be able to insert new drivers at will and to have the power of replacing an existing driver with a new version.

In order to make these things possible, DOS organizes the drivers as a singly linked chain, with a defined starting point that is always at the same place within any specific DOS version (the location differs from one version to the next, however). Each driver in the chain includes as part of its structure a pointer to the next one, and the end of the chain is signified by the value FFFFh in the offset position of the final driver's link. Unlike the MCB chain, this is a true linked list.

The original device chain is prebuilt in the hidden system file IO.SYS (in PC-DOS, IBMBIO.COM). If you add drivers to your system via the DEVICE= command in the CONFIG.SYS file, they are patched into the chain by the initialization portion of IO.SYS each time you boot your system.

Subsequent sections of this chapter describe the detailed organization of the device driver chain, tell how drivers are initialized during system boot-up, and then show you how to locate the start of the chain for any version of DOS and how to trace the driver chain and find out what is in your system.

Organization of the Device Driver Chain The device driver chain is a singly linked list structure with a defined starting point. The link itself is a far pointer (32-bit

segment:offset format) that forms the first four bytes of each device driver, and the starting point is the driver for the NUL device.

The NUL device is the "bit bucket" for both input and output; any output sent to NUL simply vanishes without trace, and any attempt to read input from this device encounters a permanent EOF condition. In itself, a device with these characteristics is handy. NUL also serves as the "anchor" location for the driver chain.

As delivered, NUL's link pointer holds the address of the supplied CON driver (the default console or keyboard/CRT routines), which is located near the front of the IO.SYS data area (which normally is at absolute address 0700h). The NUL driver however, is located near the front of the DOS data area itself, which is at a much higher address.

Because the DOS handle-processing routines know where the NUL driver is located, they can trace through the chain to locate any required driver.

As already mentioned, the DOS routines always go through the device chain, looking for a match between the name of each character device and the requested filename, when any attempt is made to open a handle for input or output. Only when no match is found in the driver chain will DOS go search the directory for a named file. This makes it impossible to either create or access a file that has the same name as any device. It might be possible to develop a form of security system based on this fact, by first creating a file and then installing a device with the same name and providing a secure method for changing the device's name during operation.

Note that only character devices have names that are used in the search; block devices are referred to by "drive letter" instead of by name. During the search, these drivers are simply skipped. Because the first match to a name ends the search, an existing driver is replaced simply by inserting the replacement into the chain where it will be encountered first and being sure that it has the same name.

How Drivers Are Initialized When you add new drivers via CONFIG.SYS, each driver is added to the front of the chain as it is encountered. This is done by copying the link values from NUL into the new driver's link and then putting the new driver's address into the NUL link instead.

Both block and character device drivers are added into the chain in the same way. Because the search always begins at the NUL driver, this guarantees that any new drivers added will be found before the built-in ones.

The pointer-patching that inserts each driver into the chain is not actually done, though, until the last step of driver installation. First, the driver's own internal initialization code is called. If an error occurs, the installation is skipped with an advisory message. If the initialization completes without error, DOS checks the driver's attribute word to determine whether the driver is for a character device or for a block device. If it's for a character device, it is added to the chain immediately.

However, if it's a block device, DOS checks the number of units installed by the initialization code; if this is zero, that signals DOS not to install the driver even though no errors were detected. Otherwise, the unit count is used to assign the next drive letter in sequence, a Disk Parameter Block (DPB; see appendix) for the device is created and filled in from information returned by the initialization process, and a Current Directory Structure (CDS; see appendix) entry for that drive letter is built, which relates the letter back to the device driver. Only after all these actions are successfully completed does the driver get patched into the chain.

The device driver specifications let you put several device drivers into a single file and specify them all by means of the single filename in the DEVICE= line. However, when you do this, you must be aware of several "gotchas" that exist. The most serious of these applies only to block devices; the code that processes CONFIG.SYS assigns memory for the Disk DPB for each such device immediately following the driver's break address. Thus, if you have more than one block device driver in the same file, all of them should return different break addresses, and these addresses should not be followed by any code that will be needed after DOS calls the driver's initialization function.

If you mix character and block device drivers in the same file (which is not prohibited by the specs but which is definitely a risky thing to do), you must be sure that all the character drivers appear in the file before any of the block drivers, for the same reason.

The best practice, of course, is to follow a rule of "one driver, one file" and thus avoid these possible problems. Sometimes, however, it may be necessary to do otherwise. When that's the case, be very careful, and if you run into strange system crashes, look closely to be sure that an errant break address pointer is not wiping out driver code.

Locating the Start of the Chain The start of the device driver chain, like that of the MCB chain discussed earlier in this chapter, can be determined using the undoc-

umented INT 21h Function 52h (Get List of Lists), described in chapter 4. The "NUL" device driver that forms the anchor point for the chain is always located in the List of Lists.

For DOS 2.x, the driver begins 17h bytes past the address returned in ES:BX by INT 21h Function 52h. With DOS 3.0, the offset is 28h, but with 3.1 that came down to 22h, and there it has remained.

The following code fragment shows how to load ES:BX with the address of the NUL driver for DOS 3.1 and up; for earlier versions, change the constant 22h to the appropriate value:

```
mov     ah, 52h    ; get List of Lists
int     21h
add     bx, 22h    ; NUL driver offset, DOS 3.1+
```

Tracing it Through Once you have located the start of the device driver chain, actual tracing through all devices (to duplicate the action of DOS during an OPEN function) is simple. The only complicating factor is the need to distinguish between character and block devices and to report block devices differently because they have no names.

The following sample program, written for MASM version 5.1 but usable with other assemblers that support the simplified segmenation directives, shows how simple it is:

```
; DEV.ASM
.model small
.stack

.data
blkdev  db      'Block: '       ; block driver message
blkcnt  db      '0 unit(s)$'

.code
dev     proc

        mov     ah,52h          ; get List of Lists
        int     21h
        mov     ax,es           ; segment to AX
        add     bx,22h          ; driver offset, 3.1 and up
        mov     di,seg blkdev
        mov     dx,offset blkdev

dev1:   mov     ds,ax
```

```
        lea     si, [bx+10]     ; step to name/units field
        test    byte ptr [bx+5], 80h    ; check driver type
        jz      dev3            ; is BLOCK driver

        mov     cx,8            ; is CHAR driver
dev2:   lodsb                   ; so output its name
IFDEF INT29
        int     29h             ; gratuitous use of undoc DOS
ELSE
        push    dx
        mov     dl, al
        mov     ah, 2           ; Character Output
        int     21h
        pop     dx
ENDIF
        loop    dev2
        jmp     short   dev4    ; then go look for next one

dev3:   lodsb                   ; get number of units
        add     al,'0'
        push    ds
        mov     ds,di
        mov     blkcnt,al       ; set into message
        mov     ah,9
        int     21h
        pop     ds

IFDEF INT29
dev4:   mov     al,13           ; send CR and LF to CRT
        int     29h             ; gratuitous use of undoc DOS
        mov     al,10
        int     29h
ELSE
dev4:   push    dx
        mov     ah,2            ; Character Output
        mov     dl,13           ; send CR and LF to CRT
        int     21h
        mov     dl,10
        int     21h
        pop     dx
ENDIF
        mov     si,bx           ; back up to front of driver
        lodsw                   ; get offset of next one
        xchg    ax,bx
        lodsw                   ; and then its segment
        cmp     bx,0FFFFh       ; was this end of chain?
        jne     dev1            ; no, loop back
```

```
        mov     ax,4C00H        ; yes, return to DOS
        int     21h

dev     endp

        end     dev
```

When DEV.EXE is run on a MS-DOS 3.3 system, it produces the following list of drivers. The bottom 12 are those contained in hidden file IO.SYS, the 3-unit block driver controls drives A:, B:, and C:, and the other 11 are the standard DOS devices:

```
NUL
4DOSSTAK
BRNDEV
CON
MS$MOUSE
Block: 1 unit(s)
XMSXXXX0
Block: 2 unit(s)
CON
AUX
PRN
CLOCK$
Block: 3 unit(s)
COM1
LPT1
LPT2
LPT3
COM2
COM3
COM4
```

The 2-unit block driver is the OnTrack disk manager required to partition an 80MB unit into three 26MB logical drives, and the single-unit block driver is Intel's QUIKMEM2.SYS RAMdisk. The duplicate name of CON is UV-ANSI.SYS; because it appears in the chain ahead of the "standard" CON driver, it is always used.

It is worth noting that, if assembled with a /DINT29 flag, DEV.ASM will make gratuitous use of undocumented DOS. INT 29h is the "fast putchar" interrupt called from DOS when sending characters to a device whose attribute word has bit 4 set. It is tempting to use INT 29h here, because it does simplify the code just below label dev2. However, chapter 1 notes that there really are places you

should use documented DOS instead of undocumented DOS, even when it seems like more trouble. Performing output in this program is one of those places. Although this program absolutely demands use of undocumented INT 21h Function 52h, there are several reasons for it *not* to use undocumented INT 29h:

- The same functionality is available with INT 21h Function 2
- Not all CON drivers support the "fast putchar" bit
- INT 29h output is not redirectable: because this program displays block devices using INT 21h Function 9, which is redirectable, using INT 29h elsewhere means that running DEV > TMP.TMP ends up displaying character devices on the screen, and block devices in the file: pretty silly!

Thus, DEV provides a nice demonstration of when undocumented DOS is needed and when it definitely isn't needed. Exercise some discretion here. Don't use undocumented DOS if you don't need to. End of lecture.

Always Double-check Your Work In chapters 1 and 2, much emphasis was laid on trying to double-check any values returned from undocumented DOS. Verification depends on redundancy, though, and we noted that it is somewhat difficult to verify undocumented DOS, because if you already had one copy of some piece of information, you probably didn't need to go to undocumented DOS to get it in the first place!

The device driver chain, however, does provide ample opportunity for double checking. Because device driver headers contain the 8-byte name of the driver, when we think we have a pointer to a device driver header, we can (and should!) check that we really do! This idea is developed in the following short sample C program, which uses in-line assembler and the _fmemcmp() function from Microsoft C 6.0:

```
/* DEVCON.C */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>

/* some device attribute bits */
#define CHAR_DEV     (1 << 15)
#define INT29        (1 << 4)
#define IS_CLOCK     (1 << 3)
#define IS_NUL       (1 << 2)
```

```
#pragma pack(1)

typedef unsigned char BYTE;

typedef struct DeviceDriver {
    struct DeviceDriver far *next;
    unsigned attr;
    unsigned strategy;
    unsigned intr;
    union {
        BYTE name[8];
        BYTE blk_cnt;
        } u;
    } DeviceDriver;

typedef struct {
    void far *dpb;
    void far *sft;
    DeviceDriver far *clock;
    DeviceDriver far *con;
    unsigned max_bytes;
    void far *disk_buff;
    void far *cds;
    void far *fcb;
    unsigned prot_fcb;
    unsigned char blk_dev;
    unsigned char lastdrv;
    DeviceDriver nul;   /* not a pointer */
    unsigned char join;
    // ...
    } ListOfLists;  // DOS 3.1+

void fail(char *s) { puts(s); exit(1); }

main(int argc, char *argv[])
{
    ListOfLists far *doslist;
    DeviceDriver far *dd;

    _asm {
        xor bx, bx
        mov es, bx
        mov ah, 52h
        int 21h
        mov doslist, bx
        mov doslist+2, es
        }
```

```
    if (! doslist)
        fail("INT 21h Function 52h not supported");
    if (_fmemcmp(doslist->nul.u.name, "NUL     ", 8) != 0)
        fail("NUL name wrong");
    if (! (doslist->nul.attr & IS_NUL))
        fail("NUL attr wrong");
    if (_fmemcmp(doslist->con->u.name, "CON     ", 8) != 0)
        fail("CON name wrong");
    if (! (doslist->con->attr & CHAR_DEV))
        fail("CON attr wrong");
    if (_fmemcmp(doslist->clock->u.name, "CLOCK$  ", 8) != 0)
        fail("CLOCK$ name wrong");
    if (! (doslist->clock->attr & IS_CLOCK))
        fail("CLOCK$ attr wrong");

    if (argv[1][0] == '-')
    {
        /* print out device chain */
        for (dd = &doslist->nul;;)
        {
            if (dd->attr & CHAR_DEV)
                printf("%.8Fs\n", dd->u.name);
            else
                printf("Block dev: %u unit(s)\n", dd->u.blk_cnt);
            if (FP_OFF(dd->next)==-1)
                break;
            dd = dd->next;
        }
    }

    /* go back to first CON driver */
    dd = &doslist->nul;
    while (_fmemcmp(dd->u.name, "CON     ", 8) != 0)
        dd = dd->next;

    /* DOS List Of Lists holds separate ptr to latest CON driver */
    puts(dd == doslist->con ? "no new CON" : "new CON");

    return 0;
}
```

This program relies heavily on the redundancy built into the DOS device chain. We can check that doslist->clock, for example, really does point to a CLOCK$ device, using both the name "CLOCK$ " and the CLOCK bit in the device attribute word. This is particularly important because the program provides structures for DOS 3.1 and higher only. If the program is run under a version of DOS where these structures are not accurate, such as DOS 2.0, 3.0, or perhaps

7.18 (though we have every expectation that these structures will be the same in that anxiously awaited version!), the program will print out an error message and then quit, rather than blindly following bogus pointers and spewing out garbage on the screen.

Because a more limited range of DOS versions is handled, the C structures really do make this program more readable than if you were using numeric offsets. It's nice to be able to say doslist.nul->next.name, for instance, and see at a glance that you're talking about the name of the device pointed to by the next field of the NUL device. The -> and . notation also makes clear that the DOS List of Lists contains the *actual* NUL header (doslist.nul), whereas it contains *pointers* to CLOCK$ (doslist->clock) and CON (doslist->con).

If you run DEVCON a dash on the command line it prints out the same list as the earlier DEV assembly-language program. Otherwise, it simply determines whether the system's current CON driver is the default CON driver located in IO.SYS (IBMBIO.COM), or whether a new one is located in front of it in the DOS device chain. This capability will be used to test out the next program.

Loading Device Drivers from the DOS Command Line

To complete what you've learned about DOS resource management, let's create a program you can use to load device drivers from the DOS command line, without having to edit CONFIG.SYS and reboot.

Ever have an MS-DOS program that required the presence of a device driver, and wish you had a way to install the driver from the command line prompt rather than having to edit your CONFIG.SYS file and then reboot the system?

Of course you can be thankful that it's so much easier to reboot MS-DOS than it is to rebuild the kernel, which is what must be done to add a device driver to UNIX. While DOS 2.x borrowed the idea of installable device drivers from UNIX, it's often forgotten that DOS in fact improved on the installation of device drivers by replacing the building of a new kernel with the simple editing of CONFIG.SYS.

Still, most of us occasionally wish we could just type a command line to load a device driver and be done with it, for truly installable device drivers.

Also, developers of device drivers often wish they had a way to debug the initialization phase of a device driver. This type of debugging usually requires a debug device driver that loads before your device driver, or it requires hard-

ware in-circuit emulation. But if you could only load device drivers *after* the normal CONFIG.SYS stage. . .

Well, wish no more. Command-line loading of MS-DOS device drivers is not only possible, it's relatively simple to accomplish once you know a little about undocumented DOS. We will now present such a program, DEVLOD, written in a combination of C and assembly language. All you have to do is type DEVLOD followed by the name of the driver to be loaded, and any parameters needed, just as you would supply them in CONFIG.SYS. For example, instead of placing the following in CONFIG.SYS:

```
device=c:\dos\ansi.sys
```

you would simply type the following on the DOS command line:

```
C:\>devlod c:\dos\ansi.sys
```

There are several ways to verify that this worked. First, you can write ANSI strings to CON and see if they are properly interpreted as ANSI commands. For example, after a DEVLOD ANSI.SYS, the following DOS command should produce a DOS prompt in reverse video:

```
C:\>prompt $e[7m$p$g$e[0m
```

On systems that don't already have a CON replacement driver, you can also use the DEVCON program just developed to verify that DEVLOD ANSI.SYS really did something:

```
C:\UNDOC\KYLE>devcon
no new CON

C:\UNDOC\KYLE>devlod \dos\ansi.sys

C:\UNDOC\KYLE>devcon
new CON
```

Finally, you can tell the new driver has been installed by running DEV and inspecting its display of the device chain: you can see your new driver at the top of the list, right after NUL, and ahead of any identically-named drivers loaded earlier:

```
C:\UNDOC\KYLE>dev
NUL
QEMM386$
...

C:\UNDOC\KYLE>devlod \dos\clock.sys

C:\UNDOC\KYLE>dev
NUL
CLOCK$
QEMM386$
...
```

DEVLOD loads both character device drivers (such as ANSI.SYS) and block device drivers (drivers that support one or more drive units, such as VDISK.SYS), whether located in .SYS or .EXE files.

How DEVLOD Works

Here is the basic structure of the DEVLOD program:

```
startup code (C0.ASM)
main (DEVLOD.C)
    Move_Loader
        movup (MOVUP.ASM)
    Load_Drvr
        INT 21h Function 4B03h (Load Overlay)
    Get_List
        INT 21h Function 52h (Get List of Lists)
        based on DOS version number:
            get number of block devices
            get value of LASTDRIVE
            get Current Directory Structure (CDS) base
            get pointer to NUL device
    Init_Drvr
        call DD init routine
            build command packet
            call Strategy
            call Interrupt
    Get_Out
        if block device:
            Put_Blk_Dev
                for each unit:
                    Next_Drive
                        get next available drive letter
```

```
                INT 21h Function 32h (Get DPB)
                INT 21h Function 53h (Translate BPB -> DPB)
                poke CDS
                link into DPB chain
Fix_DOS_Chain
    link into dev chain
release environment space
INT 21h Function 31h (TSR)
```

DEVLOD's first job is to move itself out of the way to the top of memory. This lets it load the device driver as low as possible, reducing memory fragmentation. DEVLOD loads device drivers into memory using the documented DOS function for loading overlays, INT 21h Function 4B03h. An earlier version of DEVLOD read the driver into memory using DOS file calls to open, read, and close the driver, but this made it difficult to handle .EXE driver types. By using the EXEC function instead, DOS handes both .SYS and .EXE files properly.

DEVLOD then calls our good friend, undocumented INT 21h Function 52h, to retrieve the number of block devices currently present in the system, the value of LASTDRIVE, a pointer to the DOS Current Directory Structure (CDS) array, and a pointer to the NUL device. The location of these variables within the List of Lists varies with the DOS version number.

DEVLOD requires a pointer to the NUL device because (as we saw earlier in this chapter when discussing the DEV and DEVCON programs) NUL acts as the "anchor" to the DOS device chain. Since DEVLOD's whole purpose is to add new devices into this chain, it must update this linked list.

If the DOS version indicates operation under MS-DOS 1.x, or in the OS/2 compatibility box, DEVLOD quits with an appropriate message. Otherwise, a pointer to the name field of the NUL driver is created, and the eight bytes at that location are compared to the constant "NUL " to verify that the driver is present and that the pointer is correct.

In glancing over the appendix to this book, the astute reader may have noticed an undocumented DOS function, INT 2Fh Function 122Ch, which returns in BX:AX a pointer to the header of the second device driver (NUL is first). Since DOS links together all device-driver headers, this effectively gets a pointer to the DOS driver chain. So why call INT 21h Function 52h instead?

The reason is that, like all the internal INT 2Fh AH=12h functions, INT 21h Function 122Ch was meant to be called only from DOS itself (with *all* segment registers set to DOS's kernel segment). In any case, you still need those other

variables from the List of Lists, in case you are loading a block device (which you won't know until later, *after* you've called the driver's INIT routine).

Once DEVLOD has retrieved this information, it sends the device driver an initialization packet. This is straightforward: the function Init_Drvr() forms a packet with the INIT command, calls the driver's Strategy routine, and then calls the driver's Interrupt routine. As elsewhere, DEVLOD merely mimicks what DOS does when it loads a device driver.

If the device driver INIT fails, there is naturally nothing you can do but bail out. It is important to note that you have not yet linked the driver into the DOS driver chain, so it is easy to exit if the driver INIT fails. If the driver INIT succeeds, DEVLOD can then proceed with its true mission, which takes place (oddly enough) in the function Get_Out().

It is only at this point that DEVLOD knows whether it has a block or character device driver, so it is here that DEVLOD takes special measures for block device drivers, by calling Put_Blk_Dev(). For each unit provided by the driver, that function calls undocumented DOS INT 21h Function 32h (Get DPB) and INT 21h Function 53h (Translate BPB to DPB), alters the CDS entry for the new drive, and links the new DPB into the DPB chain. These new DPBs are added after the device driver's "break address." (The BPB, DPB, and CDS are explained in detail in chapter 4 on the DOS file system.) The key point is that in Put_Blk_Dev(), DEVLOD takes information returned by a block driver's INIT routine, and produces a new DOS drive.

When loading a block device driver, DEVLOD needs a drive letter to assign to the new driver. As will be explained in great detail in chapter 4, the CDS is an undocumented array of structures, sometimes also called the Drive Info Table, which maintains the current state of each drive in the system. The array is n elements long, where n equals LASTDRIVE. DEVLOD pokes the CDS in order to install a block device driver.

The function Next_Drive() is where DEVLOD determines the drive letter to assign to a block device (if there is an available drive letter). One technique for determining the next letter, #ifdefed out within DEVLOD.C, is simply to read the "Number of Block Devices" field (nblkdrs) out of the List of Lists. However, this fails to take account of SUBSTed or network-redirected drives. Therefore, we instead walk the CDS, looking for the first free drive. In any case, DEVLOD will *update* the nblkdrs field, if it successfully loads a block device.

Whether loading a block or character driver, DEVLOD also uses the "break address" (the first byte of the driver's address space which can safely be turned back to DOS for reuse) returned by the driver. For block devices, the break address has been increased to include the newly-created DPBs. Get_Out() converts the break address into a count of paragraphs to be retained.

The function copyptr() is called three times in succession to first save the content of the NUL driver's link field, then copy it into the link field of the new driver, and finally store the far address of the new driver in the NUL driver's link field. The copyptr() function is provided in MOVUP.ASM, shown later in this chapter. Note again that the DOS linked list is not altered until after you know that the driver's INIT succeeded.

DEVLOD then links the device header into DOS's linked list of driver headers, and saves some memory by releasing its environment. (The resulting "hole in RAM" will cause no harm, contrary to popular belief. It will, in fact, be used as the environment space for any program subsequently loaded, if the size of the environment is not increased.) Finally, DEVLOD calls the documented DOS TSR function INT 21h Function 31h to exit without releasing the memory now occupied by the driver.

DEVLOD.C

Before you look at how this dynamic loader accomplishes all this in less than 2,000 bytes of executable code, some constraints should be mentioned:

Many confusing details were eliminated by implementing DEVLOD as a .COM program, using the tiny memory model of Turbo C. The way the program moves itself up in memory became much clearer when the .COM format removed the need to individually manage each segment register.

In order to move the program while it is executing, it's necessary to know every address that the program can reach during its execution. This precludes using *any* part of the libraries supplied with the compiler. Fortunately, in this case that's not a serious restriction; nearly everything can be handled without them. Two assembly-language listings take care of the few things that cannot easily be done in C itself.

Only one readily available implementation of C makes it easy to completely sever the link to the runtime libraries. That is Borland's Turbo C, which provides sample code showing how. (Microsoft also provides such a capability, but its documentation is quite cryptic.)

Thus the program, as presented, requires Turbo C with its register pseudo-variables, geninterrupt(), and __emit__() features. As explained in chapter 2, register pseudo-variables such as _AX provide a way to directly read or load the CPU registers from C. Both geninterrupt() and __emit__() simply emit bytes into the code stream; neither are actually functions.

Here is the main program, DEVLOD.C:

```
/*******************************************************************
 *     DEVLOD.C - Jim Kyle - 08/20/90                              *
 *            Copyright 1990 by Jim Kyle - All Rights Reserved     *
 *     (minor revisions by Andrew Schulman - 9/12/90)              *
 *     Dynamic loader for device drivers                           *
 *            Requires Turbo C; see DEVLOD.MAK also for ASM helpers.*
 *******************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

typedef unsigned char BYTE;

#define GETFLAGS __emit__(0x9F)
#define FIXDS    __emit__(0x16,0x1F)/* PUSH SS, POP DS              */
#define PUSH_BP  __emit__(0x55)
#define POP_BP   __emit__(0x5D)

unsigned _stklen = 0x200;
unsigned _heaplen = 0;

char FileName[65];  /* filename global buffer                      */
char * dvrarg;      /* points to char after name in cmdline buffer */
unsigned movsize;   /* number of bytes to be moved up for driver   */
void (far * driver)();  /* used as pointer to call driver code     */
void far * drvptr;  /* holds pointer to device driver              */
void far * nuldrvr; /* additional driver pointers                  */
void far * nxtdrvr;
BYTE far * nblkdrs; /* points to block device count in List of Lists*/
unsigned lastdrive; /* value of LASTDRIVE in List of Lists         */
BYTE far * CDSbase; /* base of Current Dir Structure               */
int CDSsize;        /* size of CDS element                         */
unsigned nulseg;    /* hold parts of ListOfLists pointer           */
unsigned nulofs;
unsigned LoLofs;
#pragma pack(1)
```

```
struct packet{      /* device driver's command packet                */
    BYTE hdrlen;
    BYTE unit;
    BYTE command;       /* 0 to initialize      */
    unsigned status;    /* 0x8000 is error      */
    BYTE reserv[8];
    BYTE nunits;
    unsigned brkofs;    /* break adr on return  */
    unsigned brkseg;    /* break seg on return  */
    unsigned inpofs;    /* SI on input          */
    unsigned inpseg;    /* _psp on input        */
    BYTE NextDrv;       /* next available drive */
  } CmdPkt;

typedef struct {    /* Current Directory Structure (CDS)             */
    BYTE path[0x43];
    unsigned flags;
    void far *dpb;
    unsigned start_cluster;
    unsigned long ffff;
    unsigned slash_offset;  /* offset of '\' in current path field  */
    // next for DOS4+ only
    BYTE unknown;
    void far *ifs;
    unsigned unknown2;
    } CDS;
extern unsigned _psp;       /* established by startup code in c0    */
extern unsigned _heaptop;   /* established by startup code in c0    */
extern BYTE _osmajor;       /* established by startup code      */
extern BYTE _osminor;       /* established by startup code      */

void _exit( int );          /* established by startup code in c0    */
void abort( void );         /* established by startup code in c0    */

void movup( char far *, char far *, int ); /* in MOVUP.ASM file     */
void copyptr( void far *src, void far *dst ); /* in MOVUP.ASM file  */

void exit(int c)            /* called by startup code's sequence    */
{ _exit(c);}

int Get_Driver_Name ( void )
{ char *nameptr;
  int i, j, cmdlinesz;
  nameptr = (char *)0x80;   /* check command line for driver name   */
```

```
  cmdlinesz = (unsigned)*nameptr++;
  if (cmdlinesz < 1)        /* if nothing there, return FALSE       */
    return 0;
  for (i=0; i<cmdlinesz && nameptr[i]<'!'; i++) /* skip blanks      */
    ;
  dvrarg = (char *)&nameptr[i]; /* save to put in SI                */
  for ( j=0; i<cmdlinesz && nameptr[i]>' '; i++)    /* copy name    */
    FileName[j++] = nameptr[i];
  FileName[j] = '\0';

  return 1;                 /* and return TRUE to keep going        */
}

void Put_Msg ( char *msg )
{
#ifdef INT29
    /* gratuitous use of undocumented DOS */
    while (*msg)
    { _AL = *msg++;             /* MOV AL,*msg  */
      geninterrupt(0x29);       /* INT 29h */
    }
#else
    _AH = 2;    /* doesn't need to be inside loop */
    while (*msg)
    { _DL = *msg++;
      geninterrupt(0x21);
    }
#endif
}

void Err_Halt ( char *msg )     /* print message and abort          */
{ Put_Msg ( msg );
  Put_Msg ( "\r\n" );           /* send CR,LF   */
  abort();
}

void Move_Loader ( void )       /* vacate lower part of RAM         */
{
    unsigned movsize, destseg;
    movsize = _heaptop - _psp; /* size of loader in paragraphs      */
    destseg = *(unsigned far *)MK_FP( _psp, 2 ); /* end of memory   */
    movup ( MK_FP( _psp, 0 ), MK_FP( destseg - movsize, 0 ),
            movsize << 4);      /* move and fix segregs             */
}

void Load_Drvr ( void )         /* load driver file into RAM    */
```

```
{ unsigned handle;
  struct {
    unsigned LoadSeg;
    unsigned RelocSeg;
  } ExecBlock;

  ExecBlock.LoadSeg = _psp + 0x10;
  ExecBlock.RelocSeg = _psp + 0x10;
  _DX = (unsigned)&FileName[0];
  _BX = (unsigned)&ExecBlock;
  _ES = _SS;                    /* es:bx point to ExecBlock     */
  _AX = 0x4B03;                 /* load overlay                 */
  geninterrupt ( 0x21 );        /* DS is okay on this call      */
  GETFLAGS;
  if ( _AH & 1 )
    Err_Halt ( "Unable to load driver file." );
}

void Get_List ( void )          /* set up pointers via List     */
{ _AH = 0x52;                   /* find DOS List of Lists       */
  geninterrupt ( 0x21 );
  nulseg = _ES;                 /* DOS data segment             */
  LoLofs = _BX;                 /* current drive table offset   */

  switch( _osmajor )            /* NUL adr varies with version  */
    {
    case  0:
      Err_Halt ( "Drivers not used in DOS V1." );
    case  2:
      nblkdrs = NULL;
      nulofs = LoLofs + 0x17;
      break;
    case  3:
      if (_osminor == 0)
      {
          nblkdrs = (BYTE far *) MK_FP(nulseg, LoLofs + 0x10);
          lastdrive = *((BYTE far *) MK_FP(nulseg, LoLofs + 0x1b));
          nulofs = LoLofs + 0x28;
      }
      else
      {
          nblkdrs = (BYTE far *) MK_FP(nulseg, LoLofs + 0x20);
          lastdrive = *((BYTE far *) MK_FP(nulseg, LoLofs + 0x21));
          nulofs = LoLofs + 0x22;
      }
      CDSbase = *(BYTE far * far *)MK_FP(nulseg, LoLofs + 0x16);
```

```
      CDSsize = 81;
      break;
    case  4:
    case  5:
      nblkdrs = (BYTE far *) MK_FP(nulseg, LoLofs + 0x20);
      lastdrive = *((BYTE far *) MK_FP(nulseg, LoLofs + 0x21));
      nulofs  = LoLofs + 0x22;
      CDSbase = *(BYTE far * far *) MK_FP(nulseg, LoLofs + 0x16);
      CDSsize = 88;
      break;
    case 10:
    case 20:
      Err_Halt ( "OS2 DOS Box not supported." );
    default:
      Err_Halt ( "Unknown version of DOS!");
    }
}

void Fix_DOS_Chain ( void )     /* patches driver into DOS chn  */
{ unsigned i;

  nuldrvr = MK_FP( nulseg, nulofs+0x0A ); /* verify the drvr    */
  drvptr = "NUL     ";
  for ( i=0; i<8; ++i )
    if ( *((BYTE far *)nuldrvr+i) != *((BYTE far *)drvptr+i) )
      Err_Halt ( "Failed to find NUL driver." );

  nuldrvr = MK_FP( nulseg, nulofs );    /* point to NUL driver  */
  drvptr  = MK_FP( _psp+0x10, 0 );      /* new driver's address */

  copyptr ( nuldrvr, &nxtdrvr );        /* hold old head now    */
  copyptr ( &drvptr, nuldrvr );         /* put new after NUL    */
  copyptr ( &nxtdrvr, drvptr );         /* and old after new    */
}

// returns number of next free drive, -1 if none available
int Next_Drive ( void )
{
#ifdef USE_BLKDEV
  return (nblkdrs && (*nblkdrs < lastdrive)) ? *nblkdrs : -1;
#else
  /* The following approach takes account of SUBSTed and
     network-redirector drives */
  CDS far *cds;
  int i;
  /* find first unused entry in CDS structure */
```

```
  for (i=0, cds=CDSbase; i<lastdrive; i++, ((BYTE far *)cds)+=CDSsize)
    if (! cds->flags)                /* found a free drive  */
        break;
  return (i == lastdrive) ? -1 : i;
#endif
}

int Init_Drvr ( void )
{ unsigned tmp;
#define INIT 0
  CmdPkt.command = INIT;        /* build command packet         */
  CmdPkt.hdrlen = sizeof (struct packet);
  CmdPkt.unit = 0;
  CmdPkt.inpofs  = (unsigned)dvrarg;    /* points into cmd line */
  CmdPkt.inpseg  = _psp;
  /* can't really check for next drive here, because don't yet know
     if this is a block driver or not */
  CmdPkt.NextDrv = Next_Drive();
  drvptr  = MK_FP( _psp+0x10, 0 );      /* new driver's address */

  tmp = *((unsigned far *)drvptr+3);    /* STRATEGY pointer     */
  driver = MK_FP( FP_SEG( drvptr ), tmp );
  _ES = FP_SEG( (void far *)&CmdPkt );
  _BX = FP_OFF( (void far *)&CmdPkt );
  (*driver)();                  /* set up the packet address    */
  tmp = *((unsigned far *)drvptr+4);    /* COMMAND pointer      */
  driver = MK_FP( FP_SEG( drvptr ), tmp );
  (*driver)();                  /* do the initialization        */

  /* check status code in command packet                          */
  return (! ( CmdPkt.status & 0x8000 ));
}

int  Put_Blk_Dev ( void )   /* TRUE if Block Device failed      */
{ int newdrv;
  int retval = 1;           /* pre-set for failure              */
  int unit = 0;
  BYTE far *DPBlink;
  CDS far *cds;
  int i;

  if ((Next_Drive() == -1) || CmdPkt.nunits == 0)
    return retval;          /* cannot install block driver      */
  if (CmdPkt.brkofs != 0)   /* align to next paragraph          */
  {
    CmdPkt.brkseg += (CmdPkt.brkofs >> 4) + 1;
```

```
  CmdPkt.brkofs = 0;
}
while( CmdPkt.nunits-- )
{
  if ((newdrv = Next_Drive()) == -1)
      return 1;
  (*nblkdrs)++;
  _AH = 0x32;              /* get last DPB and set poiner      */
  _DL = newdrv;
  geninterrupt ( 0x21 );
  _AX = _DS;               /* save segment to make the pointer */
  FIXDS;
  DPBlink = MK_FP(_AX, _BX);
  (unsigned) DPBlink += (_osmajor < 4 ? 24 : 25 );
  _SI = *(unsigned far *)MK_FP(CmdPkt.inpseg, CmdPkt.inpofs);
  _ES = CmdPkt.brkseg;
  _DS = CmdPkt.inpseg;
  _AH = 0x53;
  PUSH_BP;
  _BP = 0;
  geninterrupt ( 0x21 );    /* build the DPB for this unit    */
  POP_BP;
  FIXDS;
  *(void far * far *)DPBlink = MK_FP( CmdPkt.brkseg, 0 );

  /* set up the Current Directory Structure for this drive */
  cds = (CDS far *) (CDSbase + (newdrv * CDSsize));
  cds->flags = 1 << 14;       /* PHYSICAL DRIVE */
  cds->dpb = MK_FP(CmdPkt.brkseg, 0);
  cds->start_cluster = 0xFFFF;
  cds->ffff = -1L;
  cds->slash_offset = 2;
  if (_osmajor > 3)
    { cds->unknown = 0;
      cds->ifs = (void far *) 0;
      cds->unknown2 = 0;
    }

  /* set up pointers for DPB, driver  */
  DPBlink = MK_FP( CmdPkt.brkseg, 0);
  *DPBlink = newdrv;
  *(DPBlink+1) = unit++;
  if (_osmajor > 3)
    DPBlink++;           /* add one if DOS 4                 */
  *(long far *)(DPBlink+0x12) = (long)MK_FP( _psp+0x10, 0 );
  *(long far *)(DPBlink+0x18) = 0xFFFFFFFF;
  CmdPkt.brkseg += 2;       /* Leave two paragraphs for DPB   */
```

```
    CmdPkt.inpofs += 2;        /* Point to next BPB pointer      */
  }     /* end of nunits loop   */
  return 0;                /* all went okay                    */
}

void Get_Out ( void )
{ unsigned temp;

  temp = *((unsigned far *)drvptr+2);   /* attribute word       */
  if ((temp & 0x8000) == 0 )    /* if block device, set up tbls */
    if (Put_Blk_Dev() )
      Err_Halt( "Could not install block device" );

  Fix_DOS_Chain ();             /* else patch it into DOS       */

  _ES = *((unsigned *)MK_FP( _psp, 0x002C ));
  _AH = 0x49;                   /* release environment space    */
  geninterrupt ( 0x21 );

  /* then set up regs for KEEP function, and go resident        */
  temp = (CmdPkt.brkofs + 15);  /* normalize the offset         */
  temp >>= 4;
  temp += CmdPkt.brkseg;        /* add the segment address      */
  temp -= _psp;                 /* convert to paragraph count   */
  _AX = 0x3100;                 /* KEEP function of DOS         */
  _DX = (unsigned)temp;         /* paragraphs to retain         */
  geninterrupt ( 0x21 );        /* won't come back from here!   */
}

void main ( void )
{ if (!Get_Driver_Name() )
    Err_Halt ( "Device driver name required.");
  Move_Loader ();               /* move code high and jump      */
  Load_Drvr ();                 /* bring driver into freed RAM  */
  Get_List();                   /* get DOS internal variables   */
  if (Init_Drvr ())             /* let driver do its thing      */
      Get_Out();                /* check init status, go TSR    */
  else
      Err_Halt ( "Driver initialization failed." );
}
```

MOVUP.ASM

The small assembly-language module MOVUP contains two functions used in DEVLOD: movup() and copyptr(). Recall that, in order not to fragment memory,

DEVLOD moves itself up above area into which the driver will be loaded. It accomplishes this feat with movup().

The function copyptr() is located here merely because it's written in assembly language. It could have been written in C, but doing so would have required the kind of contorted expressions that have given C the reputation of being a "write-only" language. Using assembly language to transfer four bytes from source to destination makes the function much easier to understand.

```
        NAME    movup
;[]-------------------------------------------------------------[]
;|      MOVUP.ASM -- helper code for DEVLOD.C                     |
;|      Copyright 1990 by Jim Kyle - All Rights Reserved          |
;[]-------------------------------------------------------------[]

_TEXT   SEGMENT BYTE PUBLIC 'CODE'
_TEXT   ENDS

_DATA   SEGMENT WORD PUBLIC 'DATA'
_DATA   ENDS

_BSS    SEGMENT WORD PUBLIC 'BSS'
_BSS    ENDS
DGROUP  GROUP   _TEXT, _DATA, _BSS

ASSUME  CS:_TEXT, DS:DGROUP

_TEXT   SEGMENT BYTE PUBLIC 'CODE'

;-----------------------------------------------------------------
;       movup( src, dst, nbytes )
;       src and dst are far pointers. area overlap is NOT okay
;-----------------------------------------------------------------
        PUBLIC  _movup

_movup  PROC      NEAR
        push    bp
        mov     bp, sp
        push    si
        push    di
        lds     si,[bp+4]               ; source
        les     di,[bp+8]               ; destination
        mov     bx,es                   ; save dest segment
        mov     cx,[bp+12]              ; byte count
```

```
        cld
        rep     movsb                   ; move everything to high ram
        mov     ss,bx                   ; fix stack segment ASAP
        mov     ds,bx                   ; adjust DS too
        pop     di
        pop     si
        mov     sp, bp
        pop     bp
        pop     dx                      ; Get return address
        push    bx                      ; Put segment up first
        push    dx                      ; Now a far address on stack
        retf
_movup  ENDP

;-----------------------------------------------------------------
;       copyptr( src, dst )
;       src and dst are far pointers.
;       moves exactly 4 bytes from src to dst.
;-----------------------------------------------------------------
        PUBLIC  _copyptr

_copyptr        PROC    NEAR
        push    bp
        mov     bp, sp
        push    si
        push    di
        push    ds
        lds     si,[bp+4]               ; source
        les     di,[bp+8]               ; destination
        cld
        movsw
        movsw
        pop     ds
        pop     di
        pop     si
        mov     sp, bp
        pop     bp
        ret
_copyptr        ENDP

_TEXT   ENDS

        end
```

C0.ASM

Finally, startup code appears in C0.ASM, which has been extensively modified from startup code provided by Borland with Turbo C. This, or similar, code forms

part of every C program, and provides the linkage between the DOS command line and the C program itself. Normal start-up code, however, does much more than this stripped-down version: it parses the argument list, sets up pointers to the environment, and arranges things so that the signal() library functions can operate.

Since our program has no need for any of these actions, our C0.ASM module omits them. What's left just determines the DOS version in use, saving it in a pair of global variables, and trims the RAM used by the program down to the minimum. Then the module calls main(), PUSHes the returned value onto the stack, and calls exit(). Actually, if the program succeeds in loading a device driver, it will never return from main().

```
        NAME    c0
;[]-------------------------------------------------------------[]
;|      C0.ASM -- Start Up Code                                   |
;|        based on Turbo-C startup code, extensively modified     |
;[]-------------------------------------------------------------[]
_TEXT   SEGMENT BYTE PUBLIC 'CODE'

_TEXT   ENDS

_DATA   SEGMENT WORD PUBLIC 'DATA'
_DATA   ENDS

_BSS    SEGMENT WORD PUBLIC 'BSS'
_BSS    ENDS

DGROUP  GROUP   _TEXT, _DATA, _BSS

;       External References

EXTRN   _main : NEAR
EXTRN   _exit : NEAR

EXTRN   __stklen : WORD
EXTRN   __heaplen : WORD

PSPHigh         equ     00002h
PSPEnv          equ     0002ch

MINSTACK        equ     128     ; minimal stack size in words
```

```
;       At the start, DS, ES, and SS are all equal to CS

;/*-----------------------------------------------------*/
;/*      Start Up Code                                  */
;/*-----------------------------------------------------*/

_TEXT   SEGMENT BYTE PUBLIC 'CODE'

ASSUME  CS:_TEXT, DS:DGROUP

        ORG     100h

STARTX  PROC    NEAR

        mov     dx, cs          ; DX = GROUP Segment address
        mov     DGROUP@, dx
        mov     ah, 30h         ; get DOS version
        int     21h
        mov     bp, ds:[PSPHigh]; BP = Highest Memory Segment Addr
        mov     word ptr __heaptop, bp
        mov     bx, ds:[PSPEnv] ; BX = Environment Segment address
        mov     __version, ax   ; Keep major and minor version number
        mov     __psp, es       ; Keep Program Segment Prefix address

;       Determine the amount of memory that we need to keep

        mov     dx, ds          ; DX = GROUP Segment address
        sub     bp, dx          ; BP = remaining size in paragraphs
        mov     di, __stklen    ; DI = Requested stack size
;
; Make sure that the requested stack size is at least MINSTACK words.
;
        cmp     di, 2*MINSTACK  ; requested stack big enough ?
        jae     AskedStackOK    ; yes, use it
        mov     di, 2*MINSTACK  ; no,  use minimal value
        mov     __stklen, di    ; override requested stack size
AskedStackOK:
        add     di, offset DGROUP: edata
        jb      InitFailed      ; DATA segment can NOT be > 64 Kbytes
        add     di, __heaplen
        jb      InitFailed      ; DATA segment can NOT be > 64 Kbytes
        mov     cl, 4
        shr     di, cl          ; $$$ Do not destroy CL $$$
        inc     di              ; DI = DS size in paragraphs
```

```
        cmp     bp, di
        jnb     TooMuchRAM      ; Enough to run the program

;       All initialization errors arrive here

InitFailed:
        jmp     near ptr _abort

;       Set heap base and pointer

TooMuchRAM:
        mov     bx, di          ; BX = total paragraphs in DGROUP
        shl     di, cl          ; $$$ CX is still equal to 4 $$$
        add     bx, dx          ; BX = seg adr past DGROUP
        mov     __heapbase, bx
        mov     __brklvl, bx
;
;       Set the program stack down into RAM that will be kept.
;
        cli
        mov     ss, dx          ; DGROUP
        mov     sp, di          ; top of (reduced) program area
        sti

        mov     bx,__heaplen    ; set up heap top pointer
        add     bx,15
        shr     bx,cl           ; length in paragraphs
        add     bx,__heapbase
        mov     __heaptop, bx
;
;       Clear uninitialized data area to zeroes
;
        xor     ax, ax
        mov     es, cs:DGROUP@
        mov     di, offset DGROUP: bdata
        mov     cx, offset DGROUP: edata
        sub     cx, di
        rep     stosb
;
;       exit(main());
;
        call    _main           ; the real C program
        push    ax
        call    _exit           ; part of the C program too
```

```
;-----------------------------------------------------------------
;       _exit()
;       Restore interrupt vector taken during startup.
;       Exit to DOS.
;-----------------------------------------------------------------

        PUBLIC  __exit
__exit  PROC      NEAR
        push    ss
        pop     ds

;       Exit to DOS

ExitToDOS:
        mov     bp,sp
        mov     ah,4Ch
        mov     al,[bp+2]
        int     21h                     ; Exit to DOS

__exit  ENDP
STARTX  ENDP

;[]-------------------------------------------------------------[]
;|      Miscellaneous functions                                  |
;[]-------------------------------------------------------------[]

ErrorDisplay    PROC    NEAR
                mov     ah, 040h
                mov     bx, 2           ; stderr device
                int     021h
                ret
ErrorDisplay    ENDP

        PUBLIC  _abort
_abort  PROC      NEAR
                mov     cx, lgth_abortMSG
                mov     dx, offset DGROUP: abortMSG
MsgExit3        label   near
                push    ss
                pop     ds
                call    ErrorDisplay
CallExit3       label   near
                mov     ax, 3
                push    ax
                call    __exit          ; _exit(3);
```

```
_abort   ENDP

;       The DGROUP@ variable is used to reload DS with DGROUP

        PUBLIC  DGROUP@
DGROUP@     dw    ?

_TEXT   ENDS

;[]-------------------------------------------------------------[]
;|      Start Up Data Area                                       |
;[]-------------------------------------------------------------[]

_DATA   SEGMENT WORD PUBLIC 'DATA'

abortMSG        db      'Quitting program...', 13, 10
lgth_abortMSG   equ     $ - abortMSG

;
;                       Miscellaneous variables
;
        PUBLIC  __psp
        PUBLIC  __version
        PUBLIC  __osmajor
        PUBLIC  __osminor

__psp           dw      0
__version       label   word
__osmajor       db      0
__osminor       db      0

;       Memory management variables

        PUBLIC  ___heapbase
        PUBLIC  ___brklvl
        PUBLIC  ___heaptop
        PUBLIC  __heapbase
        PUBLIC  __brklvl
        PUBLIC  __heaptop

___heapbase     dw      DGROUP:edata
___brklvl       dw      DGROUP:edata
___heaptop      dw      DGROUP:edata
__heapbase      dw      0
```

```
__brklvl        dw      0
__heaptop       dw      0

_DATA   ENDS

_BSS    SEGMENT WORD PUBLIC 'BSS'

bdata   label   byte
edata   label   byte            ; mark top of used area

_BSS    ENDS

        END     STARTX
```

Make File, Plus a Brief Digression on Not Patching EXE2BIN

Since this sample program includes two assembly language modules in addition to the C source, a MAKEFILE greatly simplifies its creation. Here's one for use with Borland's MAKE utility:

```
# makefile for DEVLOD.COM - last revised 05/23/90 - jk
# can substitute other assemblers for TASM

c0.obj  :       c0.asm
        tasm c0 /t/mx/la;

movup.obj:      movup.asm
        tasm movup /t/mx/la;

devlod.obj:     devlod.c
        tcc -c -ms devlod

devlod.com:     devlod.obj c0.obj movup.obj
        tlink c0 movup devlod /c/m,devlod
        if exist devlod.com del devlod.com
        exe2bin devlod.exe devlod.com
        del devlod.exe
```

Ah, EXE2BIN: You may have some trouble here. Because Microsoft has for some time been trying to move developers away from the binary image .COM file format, and towards the more hierarchical .EXE file format, it has been difficult to find copies of the EXE2BIN utility, which attempts to convert an .EXE into

a flat binary image. Once distributed with DOS itself, EXE2BIN now comes only with the DOS *Technical Reference*.

Even if you do have a copy of EXE2BIN, you may run into the problem that it is needlessly very strict about the DOS version number. If you have EXE2BIN for DOS 3.0 and are running under DOS 3.3, for example, EXE2BIN will quit with an "Incorrect DOS Version" error message.

What to do? Several PC- and DOS-oriented magazines have published *patches* for EXE2BIN, showing how you can alter your personal copy so that it tests for a more conveninent DOS version number. This practice is extremely popular with sophisticated end-users (for example, the mammoth book *PC Magazine DOS Power Tools* contains patches for EXE2BIN and other DOS utilities).

However, the reader may have noticed that we have said next to nothing about patching DOS in this book. It is almost never necessary to patch DOS or the DOS utilities. If you have a copy of EXE2BIN.EXE that thinks it needs to run under DOS 3.0, you really don't need to smack the executable so that it will run under DOS 3.3. Instead, all you need is a tiny shell that briefly takes over the DOS Get Version Number function (INT 21h Function 30h) so that it temporarily returns a more convenient version number. Such a program uses only supported, documented DOS interfaces (particularly the supported ability to hook INT 21h itself). This is one area where you definitely don't need underhanded tricks.

The following short program, DOSVER.C, takes over INT 21h, altering the return value from Function 30h according to what you specify on the command line. It then spawns a single program. When that program calls INT 21h Function 30h, it will actually be calling the INT 21h handler in DOSVER. When the spawned program exits, DOSVER sets back the INT 21h interrupt vector, and returns to DOS:

```
/*
DOSVER.C -- set different DOS version numbers
an alternate to patching programs such as EXE2BIN
*/

#include <stdlib.h>
#include <stdio.h>
#include <process.h>
#include <dos.h>
#pragma pack(1)

void (interrupt far *old)();
```

```
unsigned dosver, old_bx, old_cx;

typedef struct {
    unsigned es,ds,di,si,bp,sp,bx,dx,cx,ax,ip,cs,flags;
    } REG_PARAMS;

void interrupt far dos(REG_PARAMS r)
{
    if ((r.ax >> 8) == 0x30)
    {
        r.ax = dosver;
        r.bx = old_bx;
        r.cx = old_cx;
    }
    else
        _chain_intr(old);
}

void fail(char *s) { puts(s); exit(1); }
main(int argc, char *argv[])
{
    int major, minor;

    if (argc < 4)
        fail("usage: dosver <major> <minor> <command...>\n\
example: dosver 3 31 exe2bin devlod.exe devlod.com");

    if (! (major = atoi(argv[1])))
        fail("bad version number");
    if ((minor = atoi(argv[2])) < 10)       /* e.g. 3.1 to 3.10 */
        minor *= 10;
    dosver = (minor << 8) + major;

    _asm mov ax, 3000h
    _asm int 21h
    _asm mov old_cx, cx                     /* OEM, serial# */
    _asm mov old_bx, bx

    old = _dos_getvect(0x21);               /* save INT 21h */
    _dos_setvect(0x21, dos);                /* hook INT 21h */
    spawnvp(P_WAIT, argv[3], &argv[3]);     /* run command */
    _dos_setvect(0x21, old);                /* unhook INT 21h */
    return 0;
}
```

If you fail to produce DEVLOD.COM with the MAKE file shown earlier, and EXE2BIN version checking is the culprit, you can substitute something like the

following line (for "3 0," substitute the DOS version that your copy of EXE2BIN thinks it needs):

```
dosver 3 0 exe2bin devlod.exe devlod.com
```

Of course, you can use DOSVER with programs other than EXE2BIN. Apparently so many programs get their DOS version number checking wrong that the next version of DOS will come with a utility, similar to DOSVER, that will let users set the DOS version number on an application-by-application basis: yuk!

How Well Does DEVLOD Work?

A fitting conclusion to this chapter is to use some of the utilities developed earlier, MEM and DEV, to see what your system looks like after you've loaded up a large number of device drivers with DEVLOD:

```
C:\UNDOC\KYLE>devlod \dos\smartdrv.sys 256 /a
Microsoft SMARTDrive Disk Cache version 3.03
    Cache size: 256K in Expanded Memory
    Room for 30 tracks of 17 sectors each
    Minimum cache size will be OK

C:\UNDOC\KYLE>devlod \dos\ramdrive.sys
Microsoft RAMDrive version 3.04 virtual disk D:
    Disk size: 64k
    Sector size: 512 bytes
    Allocation unit: 1 sectors
    Directory entries: 64

C:\UNDOC\KYLE>devlod \dos\vdisk.sys
VDISK Version 3.2 virtual disk E:
   Buffer size adjusted
   Sector size adjusted
   Directory entries adjusted
   Buffer size:          64 KB
   Sector size:         128
   Directory entries:    64

C:\UNDOC\KYLE>devlod \dos\ansi.sys

C:\UNDOC\KYLE>mem
Seg     Owner   Size            Env
09F3    0008    00F4 (  3904)         config [15 2F 4B 67 ]
0AE8    0AE9    00D3 (  3376)  0BC1   c:\dos33\command.com   [22 23 24 2E ]
0BBC    0000    0003 (    48)         free
0BC0    0AE9    0019 (   400)
0BDA    0AE9    0004 (    64)
```

```
0BDF    3074    000D (   208)
0BED    0000    0000 (     0)        free
0BEE    0BEF    0367 ( 13936)  0BE0   \msc\bin\smartdrv.sys 256 /a [13 19 ]
0F56    0F57    1059 ( 66960)  0BE0   \msc\bin\ramdrive.sys  [F1 FA ]
1FB0    1FB1    104C ( 66752)  0BE0   \dos33\vdisk.sys
2FFD    2FFE    0075 (  1872)  0BE0   \dos33\ansi.sys  [1B 29 ]
3073    3074    1218 ( 74112)  0BE0  C:\UNDOC\KYLE\MEM.EXE   [00 ]
428C    0000    7573 (481072)        free [30 F8 ]

C:\UNDOC\KYLE>dev
NUL
CON
Block: 1 unit(s)
Block: 1 unit(s)
SMARTAAR
QEMM386$
EMMXXXX0
CON
AUX
PRN
CLOCK$
Block: 3 unit(s)
COM1
LPT1
LPT2
LPT3
COM2
COM3
COM4
```

The output from MEM shows quite clearly that your device drivers really are resident in memory. Meanwhile, the output from DEV shows that they really are linked into the DOS device chain (for example, "SMARTAAR" is SMARTDRV.SYS). Of course, the real test is that, after loading SMARTDRV, RAMDRIVE, VDISK, and ANSI.SYS, my disk accesses went a bit faster (because of the new 256KB SMARTDRV disk cache in expanded memory), I had some additonal drives (created by RAMDRIVE and VDISK), and programs that assume the presence of ANSI.SYS (for shame!) suddenly started producing reasonable output. And, of course, I had a lot less memory.

One other interesting item in the MEM output is the environment segment number displayed for the four drivers. Recall that, in order to save some memory, DEVLOD releases its environment. The MEM program correctly detects that the 0BE0h environment segment still shown in the PSP for each resident instance of DEVLOD, does not in fact belong to them. The name "DEVLOD" does not precede the names of the drivers, because, as noted earlier in the discussion of MEM,

program names (which only became available in DOS 3+) are located in the environment segment, not in the PSP. Each instance of DEVLOD has jettisoned its environment, so its program name is gone too.

Who then does it belong to? Actually, it belongs to MEM.EXE itself. Since each instance of DEVLOD has released its environment, when MEM comes along there is a nice environment-sized block of free memory just waiting to be used, and MEM uses this block of memory for its environment. The reason 0BE0 shows up as an environment, not only for MEM.EXE, but for each instance of DEVLOD as well, is that when DEVLOD releases the environment, it doesn't do anything to the environment segment address at offset 2Ch in its PSP. Probably DEVLOD (and any other program that frees its environment) ought to zero out this address.

It should be noted that some device drivers appear not to be properly loaded by DEVLOD. These include some memory managers and drivers that use extended memory. For example, Microsoft's XMS driver HIMEM.SYS often crashes the system if you attempt to load it with DEVLOD. Furthermore, while DEVLOD VDISK.SYS definitely works in that a valid RAM disk is created, other programs that check for the presence of VDISK (such as protected-mode DOS extenders) often fail mysteriously when VDISK has been loaded in this unusual fashion. In the MEM display, note that the INT 19h vector is *not* pointing at VDISK.SYS as it should.

For another perspective on loading drivers, see the article by Giles Todd, "Installing MS-DOS Device Drivers from the Command Line," published in the British magazine *.EXE* (August, 1989). For background on DOS device drivers in general, two excellent books are the classic *Writing MS-DOS Device Drivers* by Robert S. Lai (Reading, MA: Addison-Wesley, 1987), and the recent *Writing DOS Device Drivers in C* by Phillip M. Adams and Clovis L. Tondo (Englewood Cliffs, NJ: Prentice Hall, 1990).

Many of the complexities of loading block devices—in particular, the importance of updating the CDS—will become clear in the next chapter, where we discuss the DOS file system.

Chapter 4

The DOS File System and Network Redirector

Jim Kyle, David Maxey, and Andrew Schulman

The file system is a truly irreplaceable part of MS-DOS. While most successful PC software bypasses many of DOS's services, and goes directly to the hardware to produce screen output or to read the keyboard, when it comes to reading and writing files, few programs spurn the DOS file system.

Actually, there are *two* DOS file systems. One, known as the FAT file system from the name of its key data structure (the File Allocation Table), is the logical structure that DOS uses for physical media such as floppy disks and hard drives. The FAT is probably the world's best-known DOS internal data structure, having entered popular culture via Peter Norton's book *Inside the IBM PC*. Along with the FAT, the other key underpinning of this file system is a structure called the Drive Parameter Block (DPB), which we will be discussing in detail later in this chapter.

The other file system, introduced in DOS 3.1, is known as the MS-DOS network redirector. It is used for mapping a DOS directory hierarchy onto "alien" (non-FAT) systems such as network file servers and CD-ROM devices, and is known as the MS-DOS network redirector. Drives created with the network redirector do not have FATs or DPBs. While networks are a tremendously important part of the DOS file system (and one that is frequently ignored in discussions

of DOS internals), the network redirector is somewhat misnamed: it isn't just for networks anymore.

All drives, whether FAT-based or non-FAT, have entries in another key DOS data structure called the Current Directory Structure (CDS) table. (The only important exceptions to this statement are drives created under Novell NetWare, which bypasses the CDS.) Many programs in this chapter manipulate the CDS in some way.

In this chapter, you will read about DOS drives, directories, and files and, like most such discussions, we will begin with physical magnetic media and work our way to the directory structure seen by a typical DOS user. However, this chapter takes a somewhat different slant, because having shown how DOS applies a logical ordering to physical media, it then proceeds to show how this same logical ordering can be applied to things other than hard drives and floppy disks. In other words, any file system is a *fiction*, and this chapter emphasizes how generic the DOS notion of a drive is: it isn't just for physical media (or even RAM disks) anymore.

This chapter contains an enormous number of sample programs, giving it more of a "cookbook" approach than other parts of the book. The chapter's *pièce de résistance* is PHANTOM.PAS, a complete example of using the DOS network redirector interface to create a new drive. Other code in this chapter includes routines to:

- Free orphaned file handles
- Derive a filename or attribute from a file handle
- Use wildcards in the DOS handle-based Rename File function
- Increase the number of program file handles before DOS 3.3
- Determine the FILES= and BUFFERS= values
- Set or turn off drive letters
- Walk the Current Directory Structure (CDS)
- Walk the System File Table (SFT)
- Get the "true" (canonical) name of a file

What ties all this together is an emphasis on the *logical* rather than the *physical* aspects of the DOS file system. But first, take a quick look at the physical aspects.

The Physical Disk: How DOS Sees It

The starting point for the FAT file system is the physical disk and the drive mechanism itself. These marvels of mechanical precision convert a stream of information represented as a sequence of bits into a corresponding sequence of magnetic flux reversals that are placed at a known location on the surface of the disk.

Entire volumes could be written on the methods by which this is done, but they would be of interest primarily to disk drive designers. As programmers, we are more interested in what has to be done to translate program-oriented descriptions of data into the form required by the actual disk hardware.

These translations occur in several layers. Our programs organize data into a stream of bytes, and store these streams into files which are later read back as streams. DOS translates our references to files into references to physical drive locations such as "drive" and "cluster," and then at a lower level converts the "cluster" reference into the more hardware-oriented values of "track," "head," and "sector" for transmission to the specified drive. The BIOS and the drive controller then translate those values into sequences of pulses that select the addressed drive, position the actuator to the desired cylinder of tracks, select the specified head, and begin reading from it when the correct sector is identified.

A concrete look at these multiple layers is provided in chapter 8 of this book, where the INTRSPY utility is used to examine in detail the process of formatting a floppy disk.

Surfaces, Tracks, and Sectors

One starting point for gaining an understanding of the DOS file system is the surface of the magnetic medium itself, as exemplified by the familiar floppy diskette (the hard disk operates in much the same way, but with much greater precision).

In the earliest days of MS-DOS, the original IBM PC came equipped with a single-head, single-sided disk drive that had a storage capacity of 160KB per disk. The head made contact with the underside of the diskette, which was placed into the drive in normal operating position, that is, on the side opposite to that on which the manufacturer's label was affixed. Balancing the pressure of the head against the lower side of the diskette was a felt pressure pad that rubbed against the upper surface.

On the single active surface, the head wrote and later read back information in one of 40 concentric tracks. The head actuator mechanism was moved in or out

to position the head accurately over the desired track. The track nearest the outer edge of the disk was designated as Track 00, and that nearest the hub hole as Track 39.

A small "index" hole near the large hub hole served as a reference point to determine disk rotation. A sensor generated an index pulse each time this hole passed over it, and since the disk rotated at a constant speed of 300 RPM (200 milliseconds per revolution), the associated controller card could measure off "sectors" around the track in which to store data. These first drives contained eight sectors per track, each sector with room for 512 bytes storage. Between sectors, an "address mark" and some special identification codes helped the controller verify that all was well with the drive.

Thus, each track contained 8*512 bytes of data, or 4,096 bytes, and the 40 tracks held a total of 163,840 bytes (which was rounded down in speech and writings to "160KB" since 1KB=1,024 bytes).

Before long, the single-sided drive was supplanted by a two-headed model that could read and write on both surfaces, immediately doubling the storage capacity to 320KB per disk. Not long after that (but before the introduction of DOS 2.0), an extra sector was added to the format, bringing the storage capacity up to the 360KB we know today.

Later, high-density 1.2MB drives, rotating at 360 RPM and holding 80 rather than 40 tracks, came along, but the basic principles hold true for them too, as for the 3.5-inch units and today's huge hard drives.

In all cases, the drive itself can only identify storage locations in terms of which head is to be used, which track (or cylinder, an alternative term) the head is to be positioned over, and which sector of that track is to be dealt with (whether reading from or writing to it).

Humans, however, have difficulty remembering a large collection of numeric values. Instead, we like to name things. It seems much simpler to remember that this text is stored in a file named "CHAP4.DOC" than that it is located at sector 14, cylinder 93, head 5, of drive 3.

That's part of what the DOS file system is all about: it permits us to deal with our programs and data as named files, and turns over to the computer the job of translating these names into the sequence of numeric data that the hardware requires. Since computers excel at dealing with numeric information, it is just another example of letting the computer do what it does best, so that the human can do likewise.

Another aspect of the DOS file system permits this type of mapping to be extended to non-storage devices. RAM disks, for example, map a directory/file structure onto fast, volatile memory. The simple I/O redirection facility provided by DOS allows you to treat the screen and keyboard (CON), serial ports (COMx) and parallel ports (LPTx) as files. "Drives" created with the DOS network redirector allow a file-system structure to be mapped onto packets that are sent over the network to another machine, possibly running a completely different file system. The file system, in other words, not only simplifies access to hardware, but also provides a *unified* form of access to otherwise disparate devices.

Logical Sector Numbers and The Cluster Concept

The first step toward simplifying the head/track/sector number sequence was to recognize that there is an alternate way of uniquely specifying every sector on a disk unit, with a single number rather than with three. The way it's done is to assign the sectors unique numbers in logical sequence. That is, the first sector of the first track under the first head (which in the fully hardware-oriented scheme would be H=0 T=00 S=1), becomes Logical Sector Number (LSN) 001. The rest of the way around that first track, on the same surface, follows in sequence, so there the LSN and the plain sector number are the same. Then, however, the LSN jumps to the other surface of the disk. For a 360KB diskette, with nine sectors per track on both sides, LSN 10 would be H=1 T=00 S=1. After all sectors on this second side are accounted for, the numbering returns H=0 T=01 S=1, which becomes LSN 19.

For other disk capacities, the exact transition points differ, but the essential point is that you can always translate a head/track/sector reference into a unique LSN—if you know how many sectors are in each track, and how many heads the disk includes. The reverse translation can be readily performed as well.

For high capacity storage units (which may contain hundreds of thousands of 512-byte sectors), the LSN is a more accurate address than DOS really needs in order to allocate disk space, and to access files. Thus, the "cluster" concept came into being.

This was actually inherited from the older CP/M operating system, though a different word ("extent") was used to describe it there. A *cluster* is simply a group of adjacent sectors that are always assigned as a unit. If a file needs only one byte, it gets a whole cluster anyway. This serves a number of purposes.

One is that it greatly reduces DOS overhead in allocating and freeing disk space, since these actions are done only a fraction as frequently as they would be if space were allocated directly in sectors. It also serves to speed up disk access by assuring that a file does not become scattered all over the drive. Even if no two clusters in the file are adjacent to each other, at least within each cluster all the sectors are together. And since "seek time" is a major part of disk I/O delay, this improves overall system performance.

One obvious disadvantage of clusters is that (when there is more than one sector per cluster) they increase the amount of disk space occupied by tiny files. For example, if there are 512 bytes/sector, and eight sectors/cluster, then the minimum space allocated to a file is 4KB, even for a file whose size in a directory listing is one byte. Not exactly a peanut cluster!

So how big is a cluster? The answer is, "it depends." Some RAMdisk programs (such as Microsoft's RAMDRIVE.SYS) actually use 1-sector clusters for space economy. Most diskette formats use a cluster of only two sectors. Hard disks for the most part use either 4-sector or 8-sector clusters. Prior to DOS 3.0, only the 8-sector cluster was used; one of the major reasons for many users to upgrade was the opportunity to reduce waste space on their disks by changing to the newer 4-sector arrangement.

To tell which clusters are used by files and which are available for assignment, DOS uses a File Allocation Table or FAT. Naturally, this structure is the backbone of the FAT file system, and if it is damaged, all data on the affected disk unit may be lost.

The FAT Structure

The FAT is always located near the front of each disk volume, immediately after the Boot Record. It may begin at what would normally be a cluster boundary, or at the first sector boundary after the Boot Record (which is always at LSN 000). Two copies of the FAT are normally maintained by DOS, in case of hard disk errors (not logical errors).

The FAT is arranged as an array of numeric values, but unlike most other numeric arrays in the MS-DOS world, each element in this array may be 12 bits long rather than 8 or 16.

Prior to DOS 3.0, all FAT entries were 12 bits in size; then an optional 16-bit size was introduced, as indicated by the top 4 bits in the highest-cluster word of

the Drive Parameter Block (DPB) (described in more detail in the appendix, and later in this chapter) to tell which size is in use for any specific volume.

Note that, even with the huge volume sizes permitted by DOS 4.x and up, the FAT element size never exceeds 16 bits (despite occasional claims to the contrary). What does increase as the volume size grows is the cluster size, and the LSN.

Each element in the FAT, whether 12 or 16 bits long, corresponds to a single cluster of the drive's storage space. The first two elements, which would refer to cluster 0 and cluster 1, instead indicate the drive's type. The first cluster number actually used is always 2.

Cluster 2 is the first one available for data; since both copies of the FAT and the volume's root directory area precede this space, the LSN for Cluster 2 must be calculated by DOS from the values provided in the DPB. From that point on, the LSN at which any cluster starts can be determined by multiplying the cluster number (minus 2) times the number of sectors per cluster, and adding the known LSN for Cluster 2. That's how DOS translates cluster numbers taken from directory entries into LSN's required by the BIOS routines that actually deal with the disk. But we're getting ahead of ourselves: DPBs are explained shortly.

The value contained in each FAT element tells whether the corresponding cluster is in use or not, and if it is, gives essential information about the file that is using it. A zero indicates that the cluster is free and can be allocated. A value of 1 or 2 never occurs. The last eight possible values (FF8h-FFFh for 12-bit FATs, or FFF8h-FFFFh for 16-bit FATs) indicate that this cluster is the last one being used by its file. (F)FF7h marks a bad cluster; (F)FF0h through (F)FF6h are reserved. Any other value indicates that the file using this cluster is continued in the cluster having that value.

Thus, the FAT forms a linked list of clusters that threads the pieces of each file together, in addition to indicating where space is available. All you have to do is determine where the very first cluster for any specified file is located, and you will be able to access everything in it. That's done by the directory structure, to which we now turn.

Directory Structure

Every disk volume (that is, each diskette in the case of drives with removeable media, or each partition in the case of those in which the media cannot be removed) has a *root directory* which is the starting point for translating human-oriented file names into system-oriented cluster numbers.

The root directory immediately follows the FAT and precedes the data storage area. Its size is established when the disk is formatted and, unlike non-root directories (which are implemented as files), can never change. A typical size for a 360KB floppy is 112 entries; for a hard disk, it's usually larger: 512 entries is typical.

Each entry in a directory, whether in the root or in a subdirectory that is reached by going through the root, consists of a 32-byte structure that contains the following information:

```
#pragma pack(1)

struct _diritem {
     char           filename[8];   /* uppercase, blank padded  */
     char           ext[3];        /* uppercase, blank padded  */
     unsigned char  attr;          /* see text for details     */
     char           unused[10];
     unsigned       ftime;
     unsigned       fdate;
     unsigned       clstr;
     unsigned long  fsize;
     };
```

In this structure, the first byte of the filename field has special significance, as does the attr byte. If the first byte of the filename is E5h, that indicates the entry refers to a file which has been ERASEd from the volume, and is free to be reused for a new file or directory entry (or possibly UNerased if you get to it in time!). If the first byte is 05h, that indicates the actual first byte value should be E5h, which is a valid character for use in a filename in DOS 3 and higher. Finally, if the first byte is 00, that indicates that neither this entry, nor any subsequent one in the directory, has ever been used. This permits searches to stop as soon as a 00 byte is found. (It also means that a stray 00 byte can make it appear as if not just one bad entry, but also all the entries following it, have disappeared from your disk.)

The attr byte indicates whether the entry refers to a file, to a subdirectory (10h), or is a volume label (08h), and, if it's a file, provides other information as well. The ftime and fdate words encode the time and date at which the file was last modified, and the fsize field indicates effective file size. Use of all these items is documented; DOS functions exist to give you their values once you access the file.

The undocumented item here, and the one we're most interested in at the moment, is the word identified as clstr; this is the cluster number for the first cluster used by the file. That's what DOS uses to translate the name to a physical address. Later in this chapter, when we trace through the process that DOS goes through in opening a file, we'll deal with clstr again.

FAKEFRMT: Initializing the FAT and Root Directory

Before going any deeper into the DOS file system's secrets, we can use what we already know to create a simple utility that will quickly erase floppy disks. It overwrites the FAT with all zeroes starting at the entry for Cluster 2 (Byte 3 of the sector for a 12-bit FAT, or byte 4 for the 16-bit version), and then writes 00 bytes over the entire root directory. Also, it rewrites the boot sector, just in case the diskette had been "bootable" before, to indicate that it is not now a system diskette.

The program, FAKEFRMT.ASM, gives the same end result you would obtain by completely reformatting the diskette, but does the job much more rapidly. (Unfortunately, it will also inadvertently bring bad tracks back into active duty.)

The program is written to deal with only a single diskette size (5.25-inch 360KB), and only one drive (A:), to keep it as simple as possible. The comments give alternate figures for using other diskette sizes or the B: drive.

While the source code is extremely simple, the final program's executable size (6KB) may shock you; that's because it includes the zeroes for *all* the sectors it writes. By letting the assembler calculate how many bytes are needed, we free ourselves from the chance of small typing errors:

```
        title   FAKEFRMT - fake format program

CODE    segment
        assume  cs:CODE, ds:CODE

        org     100h

start:  mov     dx,offset AnyKey
        mov     ah,9
        int     21h
        mov     ax,0C08h        ; wait for user keystroke
        int     21h
        xor     al,3            ; test for CTRL-C (quit)
```

```
        jz      Fini
        mov     bx,offset bootsec
        xor     dx,dx           ; logical sector number
        mov     cx,12           ; number of full sectors, 360K
                                ; change to 14 for 3.5-in 720K,
                                ; 29 for 1.2Meg, or 33 for 1.44M
        mov     al,0            ; drive code, 1 for B:
        int     26h             ; absolute sector write
        pop     bx              ; flush leftover flags word
        jnc     start           ; loop unless error
        mov     dx,offset ErrMsg
        mov     ah,9
        int     21h
Fini:   mov     ah,4Ch          ; terminate program
        int     21h

AnyKey  db      'Press any key (^C to stop)', 0Dh, 0Ah, '$'
ErrMsg  db      'Error on Drive A', 0Dh, 0Ah, '$'

bootsec:                        ; this will be the Boot Sector
        jmp     short bootsnd
        nop

; BIOS Parameter Block (BPB)
DiskName        db      'FAKEFRMT'  ; must be exactly 8 chars
BytesPerSect    dw      0200h       ; same for all diskettes
ClusterSize     db      2           ; same for all diskettes
RsrvdSect       dw      1           ; boot sec, same for all
NbrFATs         db      2           ; same for all
RootDirSize     dw      112         ; same for 720K, 224 for 1.2/1.44
TotalSectors    dw      720         ; 720K=1440, 1.2M=2400, 1.44=2880
MediaCode       db      0FDh        ; 720K=F9, 1.2M=F9, 1.44M=F0
SecPerFAT       dw      2           ; 720K=3, 1.2M=7, 1.44M=9
SecPerTrack     dw      9           ; same for 720K, 1.2=15, 1.44=18
NbrOfHeads      dw      2           ; same for all
HiddenSects     dd      0           ; same for all
NotUsed         db      11 dup (0)  ; large sectors, etc.

bootsnd:
        mov     ax,cs           ; set up segment regs
        cli                     ; Disable interrupts
        mov     ss,ax
        mov     sp,7C00h        ; where boot sec loads
        sti                     ; Enable interrupts
        mov     ds,ax
        mov     si,7C00h + msg  ; start of message
bootloop:
```

```
        lodsb
        cmp     al,0            ; at end yet?
        je      boothalt        ; yes, lock things up
        mov     ah,0Eh          ; no, send via BIOS code
        mov     bx,7
        int     10h
        jmp     short bootloop  ; and go back for next char

boothalt:
        jmp     short boothalt  ; dynamic halt

msg     equ     $ - bootsec + 0 ; calc message start
        db      'This is a DATA disk only;', 0Dh, 0Ah
        db      'Insert system disk, press any key when '
        db      'ready', 0Dh, 0Ah, 0

        org     bootsec + 510   ; skip to end of sector
        db      55h, 0AAh       ; boot sector signature

fat1    db      0FDh            ; make this match MediaCode
        db      0FFh, 0FFh
        db      509 dup(0)      ; let assembler do the calc!
        db      1*512 dup (0)   ; 2*512for 720K, 6*512 for 1.2M,
                                ; or 8*512 for 1.44 meg

fat2    db      0FDh            ; make this match MediaCode
        db      0FFh, 0FFh
        db      509 dup(0)
        db      1*512 dup (0)   ; 2*512for 720K, 6*512 for 1.2M,
                                ; or 8*512 for 1.44 meg

rootdir db      7*512 dup (0)   ; 720K same, 14*512 for others

CODE    ends

        end     start
```

The program can be assembled, linked, and turned into a .COM file using the following commands:

```
masm fakefrmt;
link fakefrmt;
exe2bin fakefrmt.exe fakefrmt.com
del fakefrmt.exe
```

As mentioned earlier, the program only formats 360KB floppies in drive A:. Each time it displays the "Press any key" message, it's ready to accept another diskette for total erasure. Hit ^C to stop.

FAKEFRMT creates an image of the first 12 sectors of a freshly-formatted diskette in memory, and then uses the documented DOS Absolute Sector Write interrupt (INT 26h) to overwrite the first 12 sectors of any diskette in Drive A with that image. To change it for use with 1.2MB 5.25-inch diskettes, or with 720KB or 1.44MB 3.5-inch units, change the numbers as indicated by the comment lines, then reassemble and relink.

Notice how, wherever possible, calculation is left to the assembler. This technique permits easy changing of such things as FAT size or the length of the root directory, and of the message written into each boot sector. (Note that FAKEFRMT's message is friendlier than the default one on PC disks.)

The 40 bytes in the program starting at DiskName are the BIOS Parameter Block (BPB); the values used are for the standard 360KB diskette, while the names indicate the meanings for each item in the block, and comments indicate the appropriate values for alternate disk sizes.

The List of Lists

Since the introduction of CONFIG.SYS with DOS version 2.0, a collection of pointers has been maintained near the start of the DOS kernel's data segment. Since the existence of this collection of pointers has never been officially documented, it's known by several names. Sometimes (for example, in Terry Dettmann and Jim Kyle's popular *DOS Programmer's Reference,* 2nd edition) it is called a "Configuration Variable Table" by analogy to minicomputer conventions, but here it is called the List of Lists since that's quite descriptive of what it actually is, and has a nice biblical ring to it as well. "List of Lists" is the name used throughout this book.

The List of Lists is a central clearinghouse for virtually all of the undocumented data concerning the DOS file system. In addition, it provides the start of the Memory Control Block chain and the device driver chain, as already discussed in chapter 3. More than any other single structure, the List of Lists is the key to reaching the undocumented areas of DOS. The following is a schematic listing of just some of the structures that can be accessed via the List of Lists:

- DOS List of Lists
- Utility functions
- Memory Control Block (MCB)
  - Program Segment Prefix (PSP)
    - Environment segment
    - File handle table
  - DOS 4.x data segment subsegment control blocks
    - STACKS segments
- Drive Parameter Block (DPB)
  - File Allocation Table (FAT)
- System File Table (SFT)
- Device driver chain
- Disk buffers
- Current Directory Structure (CDS)
  - Installable File System (IFS) record
- FCB table
- SHARE.EXE hooks
  - sharing record
  - lock record

There is a great deal of interconnection between these structures. For example, SFT entries for block devices contain a pointer to the corresponding DPB—so do disk buffers. One of the items contained in the DPB is a pointer to its corresponding device driver. Meanwhile, the heads of both the DPB and device chains are found directly in the List of Lists. Since the various undocumented structures inside DOS are interwoven with each other so tightly, you'll have to take parts of the process "in good faith" right now, without detailed explanation. That would be true no matter which of the structures we looked at first.

How the List of Lists is Arranged

The layout of the List of Lists has changed, sometimes significantly, from one version of DOS to another. Refer to the Appendix for full details. Here we emphasize only those items in the list that deal either directly or indirectly with the File System.

Prior to the introduction of network support in DOS 3.1, this area of undocumented DOS was notoriously unstable. However, once network support became available, stability was forced onto these structures, since without it, network software would not operate. Now variations between versions are (for the most

part) minor, and variations from one OEM to another in the same version are practically nonexistent. The structures reached via the List of Lists are possibly the most reliable parts of all those in undocumented DOS.

As said countless times already in this book, a far pointer to the List of Lists is returned by undocumented INT 21h Function 52h in ES:BX. In all versions to date, the actual item addressed by ES:BX is a far pointer to the first Drive Parameter Block. We identify this location as LoL+0 in the following descriptions (with LoL an unsigned char pointer, so that the offsets applied to it are always in bytes).

While DOS 4 is not as important as DOS 3, it is more convenient to discuss the DOS 4 List of Lists first, and then highlight the differences found in DOS 3. In any case, DOS 5 internals should strongly resemble those in 4, and hopefully it will be more successful than DOS 4 was, so knowledge of DOS 4 internals will have lasting benefit.

Current (DOS 4+) List Layout In DOS 4.0 and above, the important file system information kept in the List of Lists is arranged as follows:

- At LoL-8 is a far (4-byte) pointer to the currently active disk buffer.
- At LoL+0 is a far pointer to the first Drive Parameter Block. Each DPB links to the next one, forming a chain that can be used to access the DPB for any drive actually present in the system. The DPBs can also be accessed via a far pointer located in the Current Directory Structure (CDS) for the drive. Undocumented DOS function INT 21h Function 32h returns a pointer to the DPB in DS:BX; if DL contains 0 when this function is called, the DPB for the current drive will be found. If DL=1, that for Drive A is located, and so forth. In addition, undocumented INT 21h Function 1Fh does the same thing but always returns the pointer for the default drive (it sets DL to 0, then falls into the code for the more generic function). INT 21h Functions 32h and the DPB are quite important, and will be discussed in more detail below.
- At LoL+4 is a far pointer to the list of System File Tables (SFTs). These hold access information for files or devices that are accessed via handles.
- At LoL+10h is a word that contains the maximum bytes/sector of any block device in the system. Each time a block device is installed, its sector size is compared to this value, and, if larger, this value is replaced by the new maximum value.

- At LoL+12h is a far pointer to disk buffer information. The nature of this information varies depending on whether buffers are located in conventional memory or in EMS (the "/X" option to the BUFFERS= command in CONFIG.SYS).
- At LoL+16h is a far pointer to the array of Current Directory Structures (CDSs). Each drive in the system has its own CDS, which contains the path and points to the DPB for that drive. This structure also contains attribute bits that specify whether the drive exists or not, is modified by the JOIN, ASSIGN, and SUBST commands, and if it's a network drive. The CDS will be discussed in great detail later in this chapter.
- At LoL+1Ah is a far pointer to the system File Control Block (FCB) table. Remember FCBs? This table exists to permit older programs that still use FCBs instead of file handles to be used in a network situation; the FCBs in this table have a structure identical to that of the SFT entries used with handles. When a program uses FCBs, the necessary information is copied from its internal FCB to any available system FCB in this table, and the system FCB is actually used for all access. The notion of a "system FCB" is something of an oxymoron.
- At LoL+1Eh is a word that contains the number of protected FCBs (the y in the FCBS=x,y statement in CONFIG.SYS). Since the number of system FCBs is limited, while the number that may be required in a multitasking environment is not, this parameter lets you specify how many of the system FCBs must be protected against swapping when more system FCBs are requested than are actually available.
- At LoL+20h is a byte indicating the total number of block devices actually installed in the system.
- At LoL+21h is a byte that contains the value set by the LASTDRIVE command in CONFIG.SYS (the default value is 5); this value may be larger than the number of block devices actually installed, up to a maximum of 26 (LASTDRIVE=Z). Chapter 2 looked at this single byte in agonizing detail.
- At LoL+34h is a byte that shows the number of JOIN'ed drives.
- At LoL+3Bh is a far pointer to the chain of IFS (installable file system) drivers, if any are present. If the Microsoft CD-ROM Extensions (MSCDEX) are loaded under DOS 4, for instance, this field will contain a pointer to an MSCDEX device driver header such as HITACHI.SYS.

- At LoL+3Fh is a word that contains the total number of buffers (the x in the CONFIG.SYS BUFFERS x,y statement), rounded up to a multiple of 30 if the buffers are located in EMS. Following this at LoL+41h is a word that contains the number of lookahead buffers (the y in BUFFERS x,y).
- Finally, at LoL+43h is a byte that identifies the boot drive (1=A:). DOS 4 was the first version of DOS that made available the drive letter from which the system was booted. The boot drive is important knowledge for install programs, or for any program that needs to find a user's CONFIG.SYS or AUTOEXEC.BAT files.

Differences at DOS 3 With DOS 3.x, two sets of List of Lists layouts were used. The first existed only under the short-lived version 3.0; at version 3.1, when full network support was made available, this was modified significantly. We discuss the later version first (versions 3.1 through 3.3).

Sharing retry information was provided at LoL-0Ch and LoL-0Ah; the word at LoL-0Ch indicates how many times to retry an operation in case of conflict, and the word at LoL-0Ah indicates how long (in machine-dependent loops) to wait between tries.

The far pointer at LoL+12h, which points only to buffer information in DOS 4+, points to the actual buffer chain in versions 3.1 through 3.3. As shown later in this chapter, the value of BUFFERS= (available directory in 4+ but not in DOS 3) can be computed by walking the buffer chain.

The only other significant change from the layout used in version 4 and above is that none of the information at offsets above LoL+34h (the number of JOIN'd drives) exists in version 3.

The differences at version 3.0 were more major. In fact, the similarity between 3.0 and later versions stops at LoL+10h. In DOS 3.0, that location contains a single byte that indicated the number of block devices installed in the system (like the byte at LoL+20h in later versions).

At LoL+11h in DOS 3.0 is the word indicating maximum sector size in bytes, at LoL+13h the far pointer to the first disk buffer, and at LoL+17h the pointer to the CDS array. The far pointer to the "system FCB" table is at LoL+22h, and the number of protected FCBs is in the word at LoL+26h. No higher addresses are used. Neither were any offsets lower than LoL-8 used.

The Blind Alley: DOS 2 In version 2.x, DOS didn't do as much with the List of Lists. The List began at LoL-2 with the pointer to the MCB chain; none of the lower addresses found in later versions was used.

While the byte at LoL+10h, the word at LoL+11h, and the far pointer at LoL+13h all had the same meaning as they did with DOS 3.0, no CDS existed in DOS 2. Instead, the information contained in later versions in this structure (including the current directory path string) was stored in the DPB for each drive.

In addition, it's possible that any specific version of DOS in the 2.x range may vary significantly from the layout described here; when these versions were released, they were distributed only through OEM channels, and each OEM was free to modify these data structures in any way. Since none of the structures were officially documented, several OEMs did modify them. For that reason, many of our file system utilities described in this chapter are not designed to work at all with versions earlier than 3.0. That's one more reason to upgrade.

When the List is Built

As you can tell from the descriptions of the information it contains, the List of Lists is a dynamic structure that reflects any changes made to CONFIG.SYS, and even by using DOS commands that change the identity of various drives or directories while the system is running. For that reason, it is built "on the fly" each time you boot your system.

IO.SYS Initialization Code The first thing that happens when you boot your system is that the computer's "bootstrap ROM" reads the "boot sector" from the floppy disk in Drive A, if one is present. If not, it reads the boot sector from the hard drive if possible. If neither of these can be done, the system proclaims that no boot device is available and waits for you to provide one.

When the boot sector is read into RAM, at absolute address 07C00h, the code it contains then locates the first DOS hidden file (either IO.SYS or IBMBIO.COM) which always starts at Cluster 2 of the boot disk, loads it into RAM, and then transfers control to that file's initialization code.

This initialization code does many things: moves itself to the top of available memory, loads and initializes the other DOS hidden file (which sets up the DOS kernel for operation), processes the CONFIG.SYS file if one is present, uses the information contained there to build the List of Lists, and finally dispatches the primary shell, which displays the familiar "DOS prompt."

While all of these actions are undocumented, and most of them are interesting, here we should concentrate on those things that set up the List of Lists, and particularly the File System, for operation. The starting point is the processing of CONFIG.SYS, which makes any installable device drivers part of the DOS kernel, and possibly modifies certain values (for example, LASTDRIVE) that control how the List of Lists is built.

But what happens if no CONFIG.SYS file exists? Obviously no drivers will be installed, but the values that control the building of the List of Lists are assembled into IO.SYS with default values that will take effect anyway. Thus the FILES= value will be set to 8, LASTDRIVE= will be set to 5, an appropriate BUFFERS= value will be calculated from memory size and drive data, a system FCBS= value of 4,0 will take effect, and the primary shell will default to C:\COMMAND.COM /P (if C: is the boot drive).

As CONFIG.SYS is processed, any of these commands encountered will overwrite the default values. When the entire file has been parsed, with all commands executed or passed over with error messages, the List of Lists is then built from the values which then exist in DOS's CONFIG control variables. Once the list has been built, the control variables (along with the rest of the now-surplus initialization code) are discarded.

Drive Parameter Blocks For every block device (disk drive) in the system, there is a Drive Parameter Block (DPB). These 32-byte blocks contain the information that DOS uses to convert cluster numbers into Logical Sector Numbers for passing to the BIOS, and also associate the device driver for that device with its assigned drive letter. The DPBs are described in detail in the Appendix, in connection with INT 21h Function 32h.

The DPB for each drive is created immediately after DOS calls the driver's Initialize routines, during the boot process (recall in chapter 3 how the DEVLOD program loaded block device drivers: it was just mimicking DOS's operation). Those for the drivers built into IO.SYS or IBMBIO.COM (normally floppies A: and B: together with hard disk C:) are created when IO.SYS initializes itself before processing CONFIG.SYS; those for all other block devices are created as one of the final steps of installing the device's driver, while CONFIG.SYS is being processed.

To create the DPB, the code that installs drivers uses undocumented DOS function INT 21h Function 53h, passing it a far pointer to the drive's BIOS Parameter Block (BPB). A copy of that block normally is built into the device driver

itself, and the pointer is part of the information returned by the driver's Initialize routine. For the built-in disk drives, the BPBs are contained in the boot sector of each volume, and each time a volume is changed, DOS uses this BPB to rebuild the DPB in case the new volume's characteristics differ from the original values.

No DPB exists for drive letters which have no drive associated with them; the exception to this is "Drive B:" in a single-floppy system; it's always assumed to exist, even when it doesn't. Also, note that non-physical devices masquerading as drives (such as RAM disks) generally do have DPBs.

Each DPB is linked to the next one by a far pointer in the DPB structure (at offset 18h from the start of the block before DOS 4.0, and at offset 19h from DOS 4.0 on); the end of this linked list is indicated by FFFFh in the pointer's "offset" position.

While DPBs are chained together in a linked list whose root is available from the List of Lists, a better way to get the DPB for a given drive is to use the undocumented DOS Get DPB function (INT 21h Function 32h), which we saw in chapter 3 as part of DEVLOD's facility for loading block device drivers, and which is often used in disk programs such as Norton Utilities or PC Tools.

The following sample program uses INT 21h Function 32h to display capacity information for each drive on the system with a DPB. For example, on a system with a 1.2 megabyte floppy drive that had a 360KB floppy in it at the time, a 70 megabyte hard disk, and a 64KB RAMdrive (installed with DEVLOD, of course!), here is the output from the program:

```
Drive A: 512 bytes/sector * 2 sectors/cluster =
         1024 bytes/cluster * 354 clusters = 362496 bytes

Drive C: 512 bytes/sector * 8 sectors/cluster =
         4096 bytes/cluster * 17648 clusters = 72286208 bytes

Drive E: 512 bytes/sector * 1 sectors/cluster =
         512 bytes/cluster * 122 clusters = 62464 bytes
```

Actually, the program displays this information twice: once by walking the DPB linked list (whose head is at offset 0 in the List of Lists), and once by calling INT 21h Function 32h for each drive < lastdrive:

```
/*
DPBTEST.C -- uses undocumented INT 21h Function 32h (Get DPB)
    to display bytes per drive; but first walks the DPB chain,
```

```
    showing the difference between the two access methods
*/

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

#pragma pack(1)

typedef unsigned char BYTE;

typedef struct dpb {
    BYTE drive;
    BYTE unit;
    unsigned bytes_per_sect;
    BYTE sectors_per_cluster;       // plus 1
    BYTE shift;                     // for sectors per cluster
    unsigned boot_sectors;
    BYTE copies_fat;
    unsigned max_root_dir;
    unsigned first_data_sector;
    unsigned highest_cluster;
    union {
        struct {
            unsigned char sectors_per_fat;
            unsigned first_dir_sector;
            void far *device_driver;
            BYTE media_descriptor;
            BYTE access_flag;
            struct dpb far *next;
            unsigned long reserved;
            } dos3;
        struct {
            unsigned sectors_per_fat;       // WORD, not BYTE!
            unsigned first_dir_sector;
            void far *device_driver;
            BYTE media_descriptor;
            BYTE access_flag;
            struct dpb far *next;
            unsigned long reserved;
            } dos4;
        } vers;
    } DPB;

#ifndef MK_FP
#define MK_FP(seg,ofs) \
    ((void far *)(((unsigned long)(seg) << 16) | (ofs)))
```

```
#endif

void fail(char *s) { puts(s); exit(1); }

void display(DPB far *dpb)
{
    unsigned long bytes_per_clust =
        dpb->bytes_per_sect * (dpb->sectors_per_cluster + 1);

    printf("Drive %c: ", 'A' + dpb->drive);
    printf("%u bytes/sector * ", dpb->bytes_per_sect);
    printf("%u sectors/cluster = \n",
        dpb->sectors_per_cluster + 1);
    printf("         %lu bytes/cluster * ", bytes_per_clust);
    printf("%u clusters = ", dpb->highest_cluster - 1);
    printf("%lu bytes\n\n",
        bytes_per_clust * (dpb->highest_cluster - 1));
}

main()
{
    DPB far *dpb;
    union REGS r;
    struct SREGS s;

    /* floppy = single disk drive logical drive indicator 0=a 1=b */
    unsigned char far *pfloppy = (BYTE far *) 0x504L;

    int i;

#ifdef __TURBOC__
    unsigned lastdrive = setdisk(getdisk());
#else
    unsigned lastdrive;
    unsigned curdrv;
    _dos_getdrive(&curdrv);
    _dos_setdrive(curdrv, &lastdrive);
#endif

    puts("Using DPB linked list");

    s.es = r.x.bx = 0;
    r.h.ah = 0x52;
    intdosx(&r, &r, &s);
    /* pointer to first DPB at offset 0h in List of Lists */
    if (! (dpb = *((void far * far *) MK_FP(s.es, r.x.bx))))
        return 1;
```

```
    do {
        /* skip either drive A: or drive B: */
        if (((*pfloppy == 1) && (dpb->drive != 0)) ||
            ((*pfloppy == 0) && (dpb->drive != 1)))
                display(dpb);
        if (_osmajor < 4)
            dpb = dpb->vers.dos3.next;
        else
            dpb = dpb->vers.dos4.next;
    } while (FP_OFF(dpb) != -1);


    puts("Using INT 21h Function 32h");

    segread(&s);
    for (i=1; i<=lastdrive; i++)
    {
        /* skip either drive A: or drive B: */
        if ((*pfloppy == 1) && (i == 1)) continue;
        else if ((*pfloppy == 0) && (i == 2)) continue;

        r.h.ah = 0x32;
        r.h.dl = i;
        intdosx(&r, &r, &s);
        if (r.h.al != 0xFF)
            display((DPB far *) MK_FP(s.ds, r.x.bx));
    }

    return 0;
}
```

This program brings up an important reason to use INT 21h Function 32h instead of walking the DPB linked list: for removable media, INT 21h Function 32h goes to the disk, and therefore picks up the most current information. Walking the linked list, merely gets whatever DPB happens to be in memory. If you access a 360KB floppy in drive A:, put in a 1.2 megabyte floppy without accessing it, and then walk the DPB linked list, you will get the DPB for the 360KB floppy. INT 21h Function 32h would not make this mistake.

Because the DOS Get DPB function hits the disk, it is worth avoiding a read for both drives A: and B: in a system where these logical drives are mapped to the same physical floppy drive. Therefore, DPBTEST decides whether to read the DPB off of drive A: or drive B: by peeking at the logical drive indicator in the DOS low-memory data area (absolute address 0000:0504).

One last note about DPBs: many crucial DOS disk utilities were thrown into temporary confusion by the introduction of DOS 4.0, because of one byte that was changed in the DPB structure. The sectors-per-FAT field at offset 0Fh (see Appendix A) grew from a *byte* to a *word*, so all subsequent fields (including the dpb->next field) were bumped one byte as well. As noted in an extremely useful article on DPBs published at the time (Ted Mirecki, "Function 32h in DOS," *PC Tech Journal*, February 1989), this one-byte modification produced a major ripple effect in all disk utilities that relied on this undocumented DOS data structure.

System File Tables While the DPBs relate actual physical devices to the drive letters DOS uses to refer to those devices, the System File Tables (SFTs) form the backbone of the DOS file system, and have been present in DOS since version 2.0.

An SFT maintains the state of an *open* file. This includes associating a filename with a directory entry and with a physical data address, keeping track of the current position of activity within the file (the file pointer), determining file size, and maintaining the time and date stamps when a file is modified. All information contained in the directory entry for a file gets there from the SFT, and is brought back into the SFT when the file is opened.

All DOS systems have at least five SFT entries; many have 20 or more. The number of SFT entries is established by the FILES= value set in CONFIG.SYS, and defaults to eight if no such command is present. Every file handle that a program obtains from DOS by opening either a file or a device eventually leads to one of the SFTs. Later on, when we see how to develop sample programs to extract various bits of information from the SFTs, we'll see exactly how the tables are organized. Here, we take a look at how DOS uses them, first when trying to open an existing file, and then when creating a new file.

When you ask DOS to open a file, by calling the documented Open File function (INT 21h Function 3Dh), or by calling a higher-level function like fopen() which in turn calls INT 21h Function 3Dh for you, the following take place:

First, DOS searches through the handle table (also called the Job File Table, or JFT) normally located in your program's PSP to find a slot that is not currently in use, and remembers the index into the table for the first such slot that it finds. This index eventually will become the "handle" associated with the open file if all goes well; the search for a slot happens first because if no such slot is available, the file cannot be opened and there's no need to do anything more.

The chain of SFTs is now searched, looking for the first SFT entry that shows a "handle count" of zero. That indicates that the entry is available for use. If no

such entry is found, an "Out of handles" error is returned, and again the open fails.

If a free handle exists and an available SFT entry is found, DOS then accesses the root directory of the drive specified in the pathspec you passed in, by using the Current Directory Structure (CDS) for that drive. We haven't yet examined this structure in detail, but—if you're dealing with a real physical drive—DOS next extracts from the CDS a pointer to the DPB, to get the details about the drive whose root directory it will need to locate, and to perform the necessary calculations to convert cluster references into Logical Sector Numbers.

Armed with this information, the DOS routines then read the root directory into one of the DOS buffers (unless it's already there). If the supplied path contained any subdirectories, the root directory of the drive is searched, trying to match the first one specified in the supplied path. If it isn't found, the function fails with a "Path not found" error.

If this top-level directory is found, however, then the "starting cluster" value in its directory entry is converted to an LSN which is used to read that subdirectory into another buffer, or to locate the subdirectory if it's already in the buffers. This process continues until all directories in the supplied path string have been traversed, and only the filename remains to be located.

At this point, and no earlier, DOS determines whether you are dealing with a real file, or with a named (character) device such as CON or LPT1. It does so by searching the list of installed device drivers; if it finds an exact match for the filename portion of the pathspec you gave it (any extension is ignored in this test), it opens the device rather than the file. This means that all the named devices seem to exist in all directories of the file system; it also means that you cannot open any file that has the same name as one of the devices, regardless of its extension. If no matching device name is found, then the last directory is searched for the filename *and* the extension. If neither a device nor a file is found, the function fails and returns a "Path not found" error code.

If a file or device is found, the handle count in that first free SFT located earlier is set to 1, and the index of this entry in the SFT list is stored in the program's JFT. What happens after that depends on whether you're opening a file or a device.

For a file, all pertinent information from the file's directory entry is copied into the corresponding fields of the SFT entry, and the file pointer (one field of the SFT) is set to zero to indicate that nothing has yet been read from the file. For a device, certain flag bits in the SFT attribute word are set to control the input

and output functions, and a pointer to the device driver is stored in the SFT. In both situations, DOS then returns to you the handle value that it reserved at the start of the process; that value (the "magic cookie") now must be used for all references to the file. You can see this file handle is merely an index into your program's JFT, and that the byte at that index (JFT[handle]) is itself an index into the SFT.

If you're creating a new file, rather than opening an existing one, this same sequence of events occurs, with one exception—the "Path not found" error code is generated only if some directory in the path cannot be located. One significant difference occurs when the SFT entry has been filled out (before the DOS function comes back to you with a handle): the new directory entry for the just-created file is put back into the DOS buffer for that LSN, and the dirty bit for that buffer is set. This tells DOS that the buffer should be written out to disk as soon as possible and in any event before being reassigned. A directory entry is created for your new file immediately, though with a length of zero.

Each time you read from or write to the file, referencing it by means of the handle, DOS uses the supplied handle to index into your own handle table in the PSP (the JFT), and again uses the value it finds *there* to index into the SFTs. The handle is then used to perform the requested operation on the file or device referenced by its corresponding SFT entry, and the file pointer and date-time stamps in the SFT is updated accordingly. Data transfers also normally involve the DOS buffer chain, which lets DOS buffer a full sector at a time no matter how you refer to the file within your program.

When you close a file, its SFT is accessed just as for reading or writing. If the file has been written to, as indicated by a status bit in the SFT attribute word, then its directory entry is updated with information from the SFT which reflects the latest size, time, and date data. Also, any partial buffer that may not yet be written is also flushed to disk. If it hasn't been written to, these steps are skipped.

The "handle count" field of the SFT entry (the first field of each SFT, for quickest access) is decremented to reflect the fact that this handle is being disconnected from the SFT, and the SFT index in the handle table is replaced by the value FFh, which indicates an unused (and therefore available) slot.

If this was the only handle using this SFT entry, the decrement brings the handle count back to 0 and makes the entry available for reuse the next time any DOS file access is needed. DOS itself uses the SFTs when loading new processes, which means any program that accesses only one file at a time, closing it before

accessing another, thereby using only one SFT entry and leaving that entry available upon return to DOS, will leave no traces of its existence to be found later.

Some programs are not so well behaved, and keep multiple files open simultaneously. But despite many programs' insistence on having 30 to 50 files available, it's rare for the number of SFTs actually used to grow much larger than 15 to 20. Later in this chapter there is a utility that shows just what's in the SFT: usually there isn't much to see.

So far we have seen how handle-based DOS file operations manipulate SFTs. But what about FCB-based file operations? To make programs that still use the archaic FCB interface network-compatible, DOS provides "system FCBs," which were touched on briefly earlier in this chapter, and which will soon be discussed in more detail. These system FCBs are, in effect, SFT entries in disguise. The layout for both structures is the same, but a system FCB must be kept synchronized with the internal FCB that its using program believes is controlling file I/O, in addition to its normal DOS duties!

Actually, there is a close relation between SFTs and FCBs. Just as an SFT is used to maintain the state of an open file for the DOS 2+ handle functions, FCBs maintained the same state for DOS 1.x. In fact, the SFT structure is largely a superset of the old FCB structure. However, there is also an important difference: SFTs operate at the DOS level, whereas FCBs operate at the application level. Recall that FCBs reside in the application's address space, not in DOS's. The SFT in contrast allows DOS to keep tighter control over the file system. For example, whereas DOS in the days of FCBs couldn't do anything about programs that left behind open files when exiting, today's DOS uses the SFT as part of its mechanism for closing any open files when an application exits.

Of course, it's not quite correct to speak about FCBs is the past tense, as much as we would like to. The handle/SFT scheme did *not* replace FCBs, but sits alongside it. Furthermore, as we've seen, FCBs for network drives (or, in fact, for any drive created with the DOS network redirector, including the PHANTOM drive later in this chapter) have corresponding SFT entries. The interrelation between FCBs and SFTs is a good example of how the new never truly replaces the old.

Buffer Chain One of the undocumented areas of the DOS File System in which it is quite easy to "meet yourself coming back" is the buffer chain, which made its debut along with subdirectories and installable drivers in version 2.0.

The idea is simple enough: to provide a centralized buffer pool that DOS manages, thus freeing application programs from having to allocate huge buffer areas in order to get good performance.

The implementations, however, have been confused. The first time out of the gate, DOS designers chose to set up the buffer chain as a linked list of sector-sized areas, each preceded by a small header block which identified the drive currently using that buffer, the LSN of the data it contained, a status byte that indicated both what kind of data was present and whether it differed the data thatwas on the disk, and a pointer to the next buffer in the chain.

The intent was apparently to use the buffers in sequence, changing the linkages as necessary to maintain the most recently used buffers near the front of the chain. Any disk access could first scan through the chain of headers, and if it found the LSN and drive, just use the buffer content without having to read the disk again. Moving each used buffer up to the front of the chain would guarantee that any time a search reached the end of the chain without finding its sector, the buffer at the end would be the one least recently used, and thus the proper one to replace with the new data.

Unfortunately, this simple approach did not take into account the pattern by which DOS actually performs disk reads. In practice, the buffer chain filled rapidly with FAT and directory data, leaving only one or two buffers at the end to be used for *all* file data transfers.

The system was modified several times during DOS versions 2 and 3, but performance problems remained significant. When the buffer areas were redesigned again for version 4, to permit the use of expanded memory for buffers, the changes were major. The list structure went from single linkage to double linkage (forward and reverse), and hashtable lookup techniques were added. While the high-memory feature was initially flaky, the performance problems were cured. Midway through the life of version 4, the high-memory problems were also taken care of, providing both space economy and rapid retrieval.

The multiple approaches used to cure these problems preclude going into more detail about the way the buffers work. Besides, there's not much practical use for any direct dealings with them, anyhow. The most important things to know about this area are not the details of how a buffer works or how to get data from it, but to be aware of the potential pitfalls the buffering system introduces.

For instance, it's quite possible to follow all the documented rules for dealing with DOS, and still get into trouble, this can be traced directly back to the way

the buffer chain works. Here's an example: assume you have a diskette in place, which has a write protect tab on it, and you attempt to erase a file from it. The attempt will fail because of the write protect tab, but if you remove the disk, take off the tab, put it back in the drive, and reply to the "Abort/Retry/Fail?" message with "Retry," some primary shells will do only a partial delete: the FAT will reflect the deletion, but the directory entry for the file will not. This will later be reported by CHKDSK as a cross-linked file.

The apparent reason is that the buffer entry for the directory entry is discarded when you open the door to remove the write protect tab, while that for the FAT is not; to make matters worse, it is flagged to be updated as soon as possible (which means as soon as you close the door).

The cure is simple: *never* "Retry" any operation on a diskette when you have opened the drive door. Always force the operation to Fail or Abort, then start over when you have corrected the problem.

One possible use for examining the DOS buffers is merely to determine the value of BUFFERS= in a running system. In DOS 4+, this number is kept directly at LoL+3Fh, but in earlier versions there's no direct access to this value. Since it is so difficult for a running program to determine which CONFIG.SYS file was used to boot the system (prior to the availability in DOS 4 of INT 21h Function 3305h, one couldn't even tell what drive the system was booted from!), it might be useful for install, setup, and configuration programs to determine the value of BUFFERS= (and FILES=, which we'll look at later).

The following program, BUFFERS.C, merely walks the buffer chain, by counting how many buffers it finds. In DOS 4+, it directly uses the value at LoL+3Fh:

```
/* BUFFERS.C -- value of BUFFERS= */
/* also see COUNTF.C to determine value of FILES= */

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

#pragma pack(1)

#ifdef __TURBOC__
#define ASM asm
#else
#define ASM _asm
```

```
#endif

typedef unsigned char BYTE;

typedef struct dskbuf {
    struct dskbuf far *next;
    BYTE drive;
    BYTE flags;
    } DSKBUF;

unsigned buffers(void)
{
    BYTE far *doslist;

    ASM mov ah, 52h
    ASM int 21h
    ASM mov doslist+2, es
    ASM mov doslist, bx

    if (_osmajor < 4)
    {
        DSKBUF far *diskbuff;
        unsigned buffers;

        /* pointer to first disk buffer in List of Lists */
        if ((_osmajor==2) || ((_osmajor==3) && (_osminor==0)))
            diskbuff = *((void far * far *) (doslist + 0x13));
        else
            diskbuff = *((void far * far *) (doslist + 0x12));

        for (buffers=1; ; buffers++)
            if ((diskbuff = diskbuff->next) == -1L)
                break;
        return buffers;
    }
    else
        /* BUFFERS= value kept directly in List of Lists */
        return *(doslist+0x3F);
}

main()
{
    printf("BUFFERS=%d\n", buffers());
}
```

It is worth noting that the Quarterdeck Expanded Memory Manager (QEMM) comes with a program, BUFFERS.COM, which not only reports the value of BUFFERS=, but which can be used to change it on the fly. By setting a small value for BUFFERS= in CONFIG.SYS, then running LOADHI BUFFERS=xx in AUTOEXEC.BAT, your buffers can be loaded into high DOS memory, saving a little more of the precious lower 640KB for other things.

Current Directory Structure (CDS)

Another crucial undocumented DOS data structure that is reached from the List of Lists is the array of Current Directory Structures (CDS array) introduced in version 3. In DOS 2, some of the information contained in this table was kept in the DPB for each drive, but in a very different layout. The *DOS Programmer's Reference* by Terry Dettmann and Jim Kyle refers to the CDS as the "Logical Drive Table" (LDT), which is an unfortunate term since LDT already stands for Local Description Table, the name of a key data structure in protected mode on Intel 80286 and higher processors.

The CDS was adopted as part of the networking additions made to DOS, and plays a central role in manipulating foreign (not just network) file systems. Many programs later in this chapter read or manipulate the CDS.

The CDS array contains one CDS for each possible block device or drive letter on the system; that is, if you specify LASTDRIVE=Z, your system will have a 26-element CDS array, whereas if use the default value for LASTDRIVE, your CDS array will contain only five elements. Each element is 81 bytes long under DOS version 3, or 88 bytes for version 4 and up.

Turning to Appendix A, where the CDS structure is depicted in the lengthy entry for INT 21h Function 52h, we see that the CDS for each drive starts off with a 67-byte ASCIIZ string for the current path on the drive; it is from this that the CDS take its name. The following C header file presents a CDS structure; note that the term NETWORK is used to refer generically to any drive created with the MS-DOS redirector:

```
/* CURRDIR.H */

#define NETWORK     (1 << 15)
#define PHYSICAL    (1 << 14)
#define JOIN        (1 << 13)
#define SUBST       (1 << 12)
```

```
typedef unsigned char BYTE;
typedef unsigned WORD;
typedef unsigned long DWORD;
typedef BYTE far *DPB;      // provide actual DPB struct if needed

#pragma pack(1)

typedef struct {
    BYTE current_path[67];  // current path
    WORD flags;             // NETWORK, PHYSICAL, JOIN, SUBST
    DPB  far *dpb;          // pointer to Drive Parameter Block
    union {
        struct {
            WORD start_cluster; // root: 0000; never accessed: FFFFh
            DWORD unknown;
            } LOCAL;        // if (! (cds[drive].flags & NETWORK))
        struct {
            DWORD redirifs_record_ptr;
            WORD parameter;
            } NET;          // if (cds[drive].flags & NETWORK)
        } u;
    WORD backslash_offset;  // offset in current_path of '\'
    // DOS4 fields for IFS
    // 7 extra bytes...
    } CDS;
      ≡
CDS far *currdir(unsigned drive);
```

After reading through CONFIG.SYS and determining the value of LASTDRIVE=, DOS creates an array like the following (though note that in DOS 4 and higher, each CDS element is 7 bytes larger):

```
CDS cds[LASTDRIVE];
```

In a program of your own, of course, you would not create a CDS array like this. Instead, you would create a far pointer to a CDS, and assign to the pointer the value found at the appropriate offset in the List of Lists:

```
ListOfLists far *list;
CDS far *cds;
// ...
if (DOS 2.0)
    fail("no CDS");
if (DOS 3.0)
```

```
    cds = list->dos30.cds;
else
    cds = list->dos31.cds;        // DOS 3.1 and higher
```

To access the CDS for a given drive, you index into the array:

```
unsigned drive;
// ...
if (drive > lastdrive())          // see chapter 2 for lastdrive()
    fail("no such drive");        // but watch out for Novell NetWare
else
    printf("%Fs\n", cds[drive].current_path);   // print far string
```

Because the size of this array is fixed by LASTDRIVE, it normally cannot be expanded without changing CONFIG.SYS and rebooting. The space immediately above the array is occupied by the DOS kernel, and contains areas referenced by absolute pointers from many areas of DOS. However, the program LASTDRIV.COM shipped with QEMM can increase LASTDRIVE on the fly, and in order for this to mean anything, the CDS array must be expanded as well: memory is allocated for a larger CDS array, the old CDS array is copied into it, the new fields are initialized, and the CDS pointer in the List of Lists (LoL+16h, except LoL+17h in DOS 3.0) is updated to point to the new CDS array.

The first element in each drive's CDS, the current path, is not always what you might expect. The current path in the CDS tells where the data really is, rather than where you address it. We can see this in the output from a utility named ENUMDRV, the source code for which we will see shortly:

```
C:\UNDOC>\dos\join a: c:\floppy
C:\UNDOC>subst g: c:\udos
C:\UNDOC>rem e: and f: are PC/TCP (FTP Software)
C:\UNDOC>enumdrv
A    C:\FLOPPY                                          JOIN
B    B:\
C    C:\UNDOC
E    \\BIN\EXPORT\DOS                                   NETWORK
F    \\HOME\U\ANDREW                                    NETWORK
G    C:\UDOS                                            SUBST
```

Note that the COMMAND.COM prompt shows we were logged into drive C: at the \UNDOC subdirectory, and that this shows up clearly in the CDS for drive C:. The SUBST command causes data on one drive to be addressed as if it were

on another—we can see this from the CDS for drive G:. The situation is reversed when you use JOIN to refer to an entire drive as though it were a subdirectory on another drive: see the CDS for drive A: above.

In both these cases, the first byte in the current path string contains the drive letter of the SUBST or JOIN "target." But as shown in drive E: and F: in the ENUMDRV map above, this is not always the case. Attaching a drive to a network file server means that all references to the drive actually refer to the server, which is generally addressed with an opening "\\" string (that's "\\\\" from a C program). Here, using PC/TCP from FTP Software, drives E: and F: were mapped to different directories on a SUN SPARCstation running SunOS. This is a good illustration of how the DOS file system allows installable file systems (IFSs): we can use the disk of a UNIX RISC machine as though it were DOS.

As you navigate the directory tree, the path string stored here tracks your position, so that DOS can always convert your relative path references (those which do not begin with a backslash) into fully qualified path specs in order to know the search path for opening a file. In fact, the only thing that happens when you change directories (by calling INT 21h Function 3Bh, or its user-level equivalent, the CD command), is that this field gets updated.

Conversely, updating this field changes the current directory. Going into your favorite debugger, locating the CDS for the current drive, and manually editing the path string in the CDS, is sufficient to actually change directories: the change is immediately reflected in the PROMPT pg display, for example. It's now quite clear why this is called with the Current Directory Structure.

At offset 45h from the start of the CDS entry is a far pointer to the DPB; this pointer controls physical access to the drive. (This is cds[drive].dpb in the structure presented earlier.) Since the DPB in turn contains a far pointer to the actual device driver, this situation provides the linkage between the logical name of the unit, and physical access to it.

The word at offset 4Fh (the final field in the DOS 3.x CDS) contains the number of characters in the pathspec area that precede the root directory indicator. This is often initially set to a value of 2, to skip the drive letter and colon; when a SUBST command is processed, the value changes to skip not only the drive letter, but all directory names concealed by the action of SUBST. That is, SUBST G: C:\UDOS copies the string "C:\UDOS\" into the CDS entry for drive G:, copies the status word and pointers from drive C:, sets the SUBST bit in the drive G: status word, sets the directory cluster number to that for the first sector of the

UDOS directory on drive C:, and sets the word at 4Fh to a value of 7, the number of characters preceding the final "\" of the pathspec string. But since the CDS may have to store the non-DOS name of an alien file stem (such as "\\BIN\EXPORT\DOS" and "\\HOME\U\ANDREW" in the example above), the word at offset 4Fh is also used to "block off" such names.

As we proceed through this chapter, we'll see how to make use of the information contained in the CDS.

Accessing the CDS

At offset 17h in DOS 3.0 and at offset 16h in DOS 3.1 and higher, the List of Lists contains a far pointer to the CDS array. Each element in the array is 51h bytes wide in DOS 3.x and 58h bytes wide in DOS 4.x and higher, and the array contains one structure for each possible drive up to LASTDRIVE. We can package all this knowledge into a currdir() function that will be used in several programs later in this chapter. The function is called with a drive number (where drive A: is 0), and returns a far pointer to the CDS for that drive:

```
/* CURRDIR.C -- uses undocumented DOS to return pointer to
   current directory structure for a given drive */

#include <stdlib.h>
#include <dos.h>

#include "currdir.h"

typedef enum { UNKNOWN=-1, FALSE=0, TRUE=1 } OK;

CDS far *currdir(unsigned drive)
{
    /* statics to preserve state: only do init once */
    static BYTE far *dir = (BYTE far *) 0;
    static OK ok = UNKNOWN;
    static unsigned currdir_size;
    static BYTE lastdrv;

    if (ok == UNKNOWN)  /* only do init once */
    {
        unsigned drv_ofs, lastdrv_ofs;

        /* curr dir struct not available in DOS 1.x or 2.x */
        if ((ok = (_osmajor < 3) ? FALSE : TRUE) == FALSE)
            return (CDS far *) 0;
```

```
        /* compute offset of curr dir struct and LASTDRIVE in DOS
           list of lists, depending on DOS version */
        drv_ofs = (_osmajor == 3 && _osminor == 0) ? 0x17 : 0x16;
        lastdrv_ofs = (_osmajor == 3 && _osminor == 0)
                        ? 0x1b
                        : 0x21;

#if defined(__TURBOC) || (defined(_MSC_VER) && (_MSC_VER >= 600))
#ifdef __TURBOC__
#define _asm    asm
#endif

        _asm    push si    /* needs to be preserved */

        /* get DOS list of lists into ES:BX */
        _asm    mov ah, 52h
        _asm    int 21h

        /* get LASTDRIVE byte */
        _asm    mov si, lastdrv_ofs
        _asm    mov ah, byte ptr es:[bx+si]
        _asm    mov lastdrv, ah

        /* get current directory structure */
        _asm    mov si, drv_ofs
        _asm    les bx, es:[bx+si]
        _asm    mov word ptr dir+2, es
        _asm    mov word ptr dir, bx

        _asm    pop si
#else
{       // Microsoft C 5.1 -- no inline assembler available
        union REGS r;
        struct SREGS s;
        BYTE far *doslist;
        segread(&s);
        r.h.ah = 0x52;
        intdosx(&r, &r, &s);
        FP_SEG(doslist) = s.es;
        FP_OFF(doslist) = r.x.bx;
        lastdrv = doslist[lastdrv_ofs];
        dir = *((BYTE far * far *) (&doslist[drv_ofs]));
}
#endif

        /* OS/2 DOS box sets dir to FFFF:FFFF */
        if (dir == (BYTE far *) -1L) ok = FALSE;
```

```
        /* compute curr directory structure size */
        currdir_size = (_osmajor >= 4) ? 0x58 : 0x51;

    } /* end of static initializations */

    if (ok == FALSE)
        return (CDS far *) 0;

    if (drive >= lastdrv) /* is their drive < LASTDRIVE? */
        return (CDS far *) 0;

    /* return array entry corresponding to drive */
    return dir + (drive * currdir_size);
}
```

Like most of the LASTDRV programs in chapter 2, currdir() uses offsets computed at run time, rather than C data structures set at compile time, because this seems better suited to the volatility of undocumented DOS. The assumption is that DOS 5.0 and higher will be fairly compatible with DOS 4.0. There are problems with this assumption, though, since while the DOS box in OS/2 1.10 presents itself to a program as DOS 10.10, in fact it more closely resembles DOS 3.x than DOS 4.x. The test for (_osmajor >= 4) incorrectly groups the OS/2 DOS box together with DOS 4.x instead of DOS 3.x. However, the DOS boxes in OS/2 1.x and 2.x don't provide a CDS anyway, and the currdir() is prepared for this possibility by checking for the invalid -1 pointer (FFFF:FFFF).

Walking the CDS

The currdir() function goes to some trouble to ensure that it can be called frequently without a lot of duplicated effort. This way, currdir() can be called in a loop for each drive in the system, producing the ENUMDRV output shown earlier. Here is ENUMDRV.C, which contains the currdir() loop:

```
/*
ENUMDRV.C -- uses currdir() in CURRDIR.C
*/

#include <stdlib.h>
#include <stdio.h>

#include "currdir.h"

void fail(char *s) { puts(s); exit(1); }
```

```
main()
{
    CDS far *dir;
    int i;
#ifdef __TURBOC__
    int lastdrv = setdisk(getdisk());
#else
    int currdrv, lastdrv;
    _dos_getdrive(&currdrv);
    _dos_setdrive(currdrv, &lastdrv);
#endif
    for (i=0; i<lastdrv; i++)
        if (! (dir = currdir(i)))
            fail("can't get current directory structure");
        else if (dir->flags)    /* is this a valid drive? */
        {
            printf("%c\t%-50Fs", 'A' + i, dir->current_path);
            if (dir->flags & NETWORK) printf("NETWORK ");
            if (dir->flags & JOIN) printf("JOIN ");
            if (dir->flags & SUBST) printf("SUBST");
            putchar('\n');
        }
    return 0;
}
```

This program will not show Novell network drives that are mapped to drive numbers greater than that specified by LASTDRIVE. As noted in chapter 2, Novell NetWare drives by default *start* at the value *after* LASTDRIVE. These drives are not found in the CDS, and are the one important exception to our statement that the CDS ties together all DOS drives. Novell doesn't use the CDS because it provides network services by hooking INT 21h, rather than by using the redirector. The reason for this is Novell has been providing DOS networking since 2.x, before there was a redirector or a CDS.

Finding the True Name of a File

In the MS-DOS file system, things may not be what they seem: a file called D:\FOO.BAR may actually be located on the floppy disk in drive A:, and a subdirectory called F:\SOURCES may actually be located on a network file server (probably not even running MS-DOS), in a directory called \\BIN\EXPORT\DOS. A "canonical" (true) path string resolves all these "logical" (that is, non-physical) drive and path references to an absolute pathname, taking account of any renaming due to JOIN, SUBST, ASSIGN, or network redirections.

Clearly, the CDS could be used to determine the canonical name of a file. One would replace the root of the filename with root of the CDS pathname string. In the SUBST example used earlier, the "true" name of G:\FOO.BAR would be C:\UDOS\FOO.BAR. But sometimes these manipulations are more complicated, as in the case of JOINed drives or filenames with complex paths.

Fortunately, there is a DOS function that will provide the true canonical form of a filename: undocumented INT 21h Function 60h (Resolve Path String to Canonical Path String). This corresponds to the undocumented TRUENAME command in COMMAND.COM in DOS 4.01, mentioned in chapter 6. Function 60h and the TRUENAME command relate to the CDS in that, for any drive n:, the output of Function 60h with the string "n:." (that is, with the . subdirectory) is identical to cds[drive n:]->current_path. Actually, there is one difference between using Function 60h and using the CDS: Function 60h hits the disk.

We can wrap INT 21h Function 60h into a TRUENAME utility. Sometimes it is useful to know the absolute pathname of a file, or to have some assistance from the operating system in interpreting complex pathnames with many .. and . subdirectories (in DOS 4 and higher, type .\TRUENAME to avoid using the undocumented TRUENAME internal command of COMMAND.COM):

```
C:\UNDOC>  truename foo\bar\..\..\.
C:\UNDOC
C:\UNDOC>  subst d: c:\undoc
C:\UNDOC>  truename d:\truename.exe
C:\UNDOC\TRUENAME.EXE
C:\UNDOC>  truename f:\
\\HOME\U\ANDREW
C:\UNDOC>  truename e:\*.exe
\\BIN\EXPORT\DOS\????????.EXE
```

TRUENAME.C consists of only a few lines of code. Note, however, that Function 60h only became available in DOS 3.0, which was the version that added the CDS to DOS. The function is also supported in the OS/2 DOS box, so TRUENAME operates correctly in that environment as well, even though there's no CDS.

```
/* TRUENAME.C */

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <dos.h>
```

```
#ifdef __TURBOC__
#define _asm asm
#endif

void ret(char *s, int retval) { puts(s); exit(retval); }

char far *truename(char far *s, char far *d)
{
    char far *s2;

    /* INT 21h AH=60h doesn't like leading or trailing blanks */
    while (isspace(*s))
        s++;
    s2 = s;
    while (*s2) s2++;
    s2--;
    while (isspace(*s2))
        *s2-- = 0;

    _asm push di
    _asm push si
    _asm les di, d
    _asm lds si, s
    _asm mov ah, 60h
    _asm int 21h
    _asm pop si
    _asm pop di
    _asm jc error
    return d;
error:
    return (char far *) 0;
}

main(int argc, char *argv[])
{
    char buf[128];
    char far *s;
    if (argc < 2)
        ret("usage: dospath <filename>", 1);
    if (_osmajor < 3)
        ret("requires DOS 3.0 or greater", 1);

    if (s = truename(argv[1], buf))
        ret(s, 0);
    else
        ret("invalid filename", 1);
}
```

This program can be compiled with either Microsoft C 6.0 or Turbo C, and uses a preprocessor definition to work with the two compilers' slightly different varieties of in-line assembler.

The truename() function can also be used as a way of double-checking our access to the CDS. For example, in ENUMDRV.C we could add the following test (which uses the _fstrcmp() function from Microsoft C 6.0) just before printing out dir->current_path:

```
if (i > 1)  /* don't hit drive A: or B: */
{
    char s[4], buf[128];
    s[0] = 'A' + i; s[1] = ':'; s[2] = '.'; s[3] = '\0';
    if (_fstrcmp(dir->current_path, truename(s, buf)) != 0)
        fail("something wrong");
}
```

System FCBs

We've already examined the SFTs used for control of both files and devices opened with the newer handle-oriented functions that were introduced into DOS 2.0. But before DOS 2.0, there were File Control Blocks (FCBs), inherited from the CP/M operating system that had been the standard of the 8-bit microcomputer world. As noted earlier, FCBs reside in an application's address space.

When the SFTs were introduced, it was necessary to retain the ability to use FCBs, for compatibility with all existing MS-DOS programs at the time. Even today, some DOS functionality *still* uses FCBs: COMMAND.COM uses them for the DEL and REN commands, and extended FCBs are necessary to change volume labels. New technology rarely replaces old techniques, so the persistence of FCBs is not surprising. At the same time, it didn't make a lot of sense to keep two totally different systems for dealing with files. The solution to this quandary was to introduce "system" FCBs, maintained by DOS itself—just as the SFTs are.

A system FCB looks exactly like an SFT entry; the layout of the structure is identical. Thus, all the internal DOS routines that get information from, or write data into, an SFT entry will work exactly the same way on a system FCB. The only difference is that the system FCBs form a separate list from the SFTs.

For unknown reasons, DOS fills the system FCBs with the character 'A' when they are built, except for those few bytes that contain validity-check flags.

Programs using the FCB approach have no knowledge of system FCBs. Since the programs cannot deal directly with the system FCBs, a group of procedures inside DOS copy data from the user's own FCB into a system FCB, and back again. When a program uses one of the older FCB DOS functions, and passes the address of the FCB to DOS as a part of the required arguments to that function, DOS then copies all pertinent information from the program's own FCB into a system FCB, does the actual work using the system FCB, and finally copies the information back into the user's original FCB before returning control to the user program.

While this may sound like unnecessary added work, it does permit programs that use the older calls to coexist with the newer techniques. The SFT/system-FCB approach is needed to provide the degree of control necessary to deal with file sharing, networking and multiple users, while still allowing ancient programs to run without change.

System File Tables (SFTs) and Job File Table (JFT)

We earlier looked at the file tables maintained in DOS, but it is now time to build some useful programs with these tables. As mentioned, the SFTs are a linked list of system-wide tables that maintain state for all open files in all processes, and the JFT is the Job File Table, found inside a PSP, that contains that process' open file handles. In the following section, we'll see three small programs that examine these tables. Two programs walk through the SFTs, and one relates a program's file handles to the information kept in the SFTs.

How Many FILES?

Earlier, we walked through the DOS disk buffers to determine the value of BUFFERS=; we can likewise walk through the SFTs to determine the value of FILES=. Just as with BUFFERS=, this value is normally set in CONFIG.SYS, though it can be altered on the fly with a utility like Quarterdeck's FILES.COM.

Our program, COUNTF, doesn't change FILES=, but merely determines its current value by threading through the SFT headers and keeping count of the number of entries in each table. The first SFT appears to always hold five possible open-file entries. If FILES=40 appears in CONFIG.SYS, for example, then DOS allocates a second SFT, large enough for 35 more files, and chains it to the first SFT. Since each header consists of a count of the number in its associated

table, together with a pointer to the next header, it's easy to count the number of possible files:

```
/*****************************************************************
 *     COUNTF.C - Jim Kyle
 *  Last change : 13 August 1990
 *****************************************************************/
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>

#ifndef MK_FP
#define MK_FP( s, o ) (void far *)(((long)(s) << 16) | \
  (unsigned)(o))
#endif

unsigned files(void)
{ union REGS regs;
  struct SREGS sregs;
  unsigned int far *ptr;
  int n = 0;

  if ( _osmajor < 2 )
    { puts( "FILES not supported in this DOS version.");
      exit(0);
    }
  regs.h.ah = 0x52;
  intdosx( &regs, &regs, &sregs );
  ptr = (unsigned int far *) MK_FP( sregs.es, regs.x.bx + 4 );
  ptr = (unsigned int far *) MK_FP( ptr[1], ptr[0] );
  while ( FP_OFF(ptr) != 0xFFFF )
    { n += ptr[2];
      ptr = (unsigned int far *) MK_FP( ptr[1], ptr[0] );
    }
  return n;
}

#ifdef TESTING
void main( void )
{   printf("FILES=%d\n", files());
}
#endif
```

It's just a matter of chaining through the blocks, accumulating the number of entries in each, until reaching the end of the chain. Even if some SFTs have been

loaded into high memory by one of the newest memory manager programs such as 386MAX or QEMM, COUNTF.C will find them.

What Files Are Now Open?

Having FILES=25 does not, of course, mean you necessarily have 25 files open at the same time on your system. On the other hand, even if each process you launch retains the DOS default maximum-handles value of 20 (a value we will see altered later in this chapter), it's possible with enough SFTs available to have 40 or even 60 files open at the same time.

Our next utility displays information about all open files and devices on your system. For instance, here is the output from FILES when running in a DOS box in Windows 3.0:

```
C:\WIN30>files > files.log
C:\WIN30>type files.log
Filename                Size      Attr Handles    Owner
--------                ----      ---- -------    -----
AUX     .                  0      0000    14      9DA8    [NOT PSP]
CON     .                  0      0000    44      9DA8    [NOT PSP]
PRN     .                  0      0000    14      9DA8    [NOT PSP]
WIN386  .SWP          999424      0020    1       4138
USER    .EXE          231680      0020    1       421A
COURE   .FON           21360      0020    1       421A
HELVE   .FON           59696      0020    1       421A
PROGMAN .EXE           55200      0020    1       421A
PROGMAN .EXE           55200      0020    1       421A
VGAOEM  .FON            5584      0020    1       421A
EGA80WOA.FON            5680      0020    1       421A
EGA40WOA.FON            8736      0020    1       421A
CGA80WOA.FON            4672      0020    1       421A
CGA40WOA.FON            6704      0020    1       421A
PIFEDIT .EXE           40124      0020    1       421A
VGAFIX  .FON            5776      0020    1       421A
WINOA386.MOD           29520      0020    1       421A
COMM    .DRV            7088      0020    1       421A
GDI     .EXE          129691      0020    1       421A
FILES   .LOG             524      0020    2       421A
```

Normally there aren't nearly this many open files. Often you must redirect FILES's output to a file (for example, files > files.log) to see anything other than AUX, CON, and PRN. When its output is redirected, FILES inherits an open file

from COMMAND.COM: this shows up in the last line in the listing above as FILES.LOG, with two owners.

Just as COUNTF did, FILES walks the SFTs. However, FILES descends into each SFT to get its information. The FILES program starts with the first SFT, pointed to by the DOS List of Lists, displays any files in that table, and then goes into a loop following the sft->next field, until it finds a next field whose segment is zero or whose offset is -1 (FFFFh).

Under DOS 3.0 and higher, the FILES program puts a lot of effort into finding file oddities, such as files whose owner is not a legitimate PSP, and file handles which have been "orphaned." For example:

```
C:\UNDOC>  \undoc\rmichels\tsrfile > nul
C:\UNDOC>  files > tmp.tmp
C:\UNDOC>  type tmp.tmp
Filename                Size      Attr Handles    Owner
--------                ----      ---- -------    -----
AUX     .                  0      0000    8       9DED    [NOT PSP]
CON     .                  0      0000    22      9DED    [NOT PSP]
PRN     .                  0      0000    8       9DED    [NOT PSP]
NUL     .                  0      0000    1       0AE9    [ORPHAN]
TMP     .TMP               0      0020    2       0AE9
```

The first three entries—AUX, CON, and PRN—are always present in the first SFT. FILES prints out [NOT PSP] after the owner ID 9DEDh because it determined that this was not a legitimate PSP. Instead, the value is apparently the effective PSP at the time that the SYSINIT initialization code in IBMBIO.COM or IO.SYS opens them (SYSINIT relocates itself to the top of memory, accounting for the high address).

The next entry displayed above, NUL, is marked as an [ORPHAN]. An "orphaned" file handle is generally the result of redirecting the output from a memory-resident utility to a file, as done here with one of the TSRs from chapter 5 of this book. TSR>NUL leaves behind an open SFT entry for NUL because DOS can't close a process' files when it terminates via the TSR call (INT 21h Function 31h). Such orphaned file handles can cause mysterious system crashes because, with enough orphans clogging up the SFTs, there would be no free entries left to open files, and many programs unfortunately blithely assume that all their file opens are successful.

In the example above, FILES determined that NUL was an orphan because its owner was COMMAND.COM, yet it had only one owner. The program gets the PSP for COMMAND.COM (using an almost 100% reliable technique from chapter 6 of this book) and compares this with the owner PSP. If a file's owner is COMMAND.COM, it might be an orphan. In this example, TMP.TMP (to which the output of FILES was redirected) was *not* an orphan. But NUL has only one owner, and that owner is COMMAND.COM. This is a sure tip-off that the other party in the redirection hasn't exited. Since the TSR has no possible use for this NUL handle (which it doesn't even know about), it is safe to close this handle: this will be done later in this chapter, in the FREEUP program.

```
/* FILES.C -- list all files in system file table */
/* see the heavily revised version on disk in CHAP4\FILES.C */
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

#ifdef __TURBOC__
#define ASM asm
#else
#define ASM _asm
#endif

typedef unsigned char BYTE;
typedef unsigned USHORT;
typedef unsigned long ULONG;
typedef BYTE far *FP;

#pragma pack(1)

typedef struct file {
    USHORT num_handles, open_mode;
    BYTE fattr;
    USHORT dev_info;    // includes drive number
    FP ptr;
    USHORT start_cluster, time, date;
    ULONG fsize, offset;
    USHORT rel_cluster, abs_cluster, dir_sector;
    BYTE dir_entry;    //USHORT dir-entry for DOS 3.0!
    BYTE filename[11];
    ULONG share_prev_sft;
    USHORT share_net_machine;
    USHORT owner_psp;
    // ...
```

```
    } file; // for DOS 3.1 and higher

typedef struct sysftab {
    struct sysftab far *next;
    USHORT num_files;
    file f[1];
    } SYS_FTAB;

typedef struct {
    BYTE type;
    USHORT owner;    /* PSP of the owner */
    USHORT size;
    BYTE unused[3];
    BYTE dos4[8];
    } MCB;

void fail(char *s) { puts(s); exit(1); }

#ifdef __TURBOC__
#define GETVECT(intno)  getvect(intno)
#define ASM             asm
#else
#define GETVECT(intno)  _dos_getvect(intno)
#define ASM             _asm
#endif

#ifndef MK_FP
#define MK_FP(seg,ofs)  ((FP)(((ULONG)(seg) << 16) | (ofs)))
#endif

#define NEXT(mcb)       (MK_FP(FP_SEG(mcb) + (mcb)->size + 1, 0))

int belongs(FP vec, USHORT start, USHORT size)
{
    USHORT seg = FP_SEG(vec);
    return (seg >= start) && (seg <= (start + size));
}

int is_psp(USHORT seg)
{
    return ((((MCB far *) MK_FP(seg-1,0))->owner == seg) &&
            (*((USHORT far *) MK_FP(seg,0)) == 0x20CD));
}

/*
    Look for "orphaned" file handles: e.g., TSR>FOO.BAR or TSR>NUL
    will leave FOO.BAR or NUL entry in SFT, consuming file handle. If
```

```
    the PSP of the file's owner is COMMAND.COM, and if there's only
    one owner, then we decide it's an orphaned handle.
*/
int orphan(file far *ff)
{
    static command_com_psp = 0;
    if (! ff->num_handles)
        return 0;
    if (! command_com_psp)  /* do just one time */
    {
        FP int2e = (FP) GETVECT(0x2E);
        MCB far *mcb;
        ASM mov ah, 52h
        ASM int 21h
        ASM mov ax, es:[bx-2]
        ASM mov word ptr mcb+2, ax
        ASM mov word ptr mcb, 0
        while (mcb->type != 'Z')
            if (belongs(int2e, FP_SEG(mcb), mcb->size))
            {
                command_com_psp = mcb->owner;
                break;
            }
            else
                mcb = (MCB far *) NEXT(mcb);
    }
    return ((ff->owner_psp == command_com_psp) &&
            (ff->num_handles == 1));
}

#define IS_AUX(s)   ((s[0]=='A') && (s[1]=='U') && (s[2]=='X'))
#define IS_CON(s)   ((s[0]=='C') && (s[1]=='O') && (s[2]=='N'))
#define IS_PRN(s)   ((s[0]=='P') && (s[1]=='R') && (s[2]=='N'))

main(void)
{
    SYS_FTAB far *sys_filetab;
    file far *ff;
    int size;
    int i;

    ASM mov ah, 52h
    ASM int 21h
    ASM les bx, dword ptr es:[bx+4] /* ptr to list of DOS file tables */
    ASM mov word ptr sys_filetab, bx
    ASM mov word ptr sys_filetab+2, es
```

```
/* DOS box of OS/2 1.x doesn't provide system file tbl */
if (sys_filetab == (SYS_FTAB far *) -1L)
    fail("system file table not supported");

switch (_osmajor)
{
    case 2:             size = 0x28; break;
    case 3:             size = 0x35; break;
    default:            size = 0x3b; break;
}

/* Perform sanity check: determine size of file structure
   empirically from difference between strings "CON" and
   "AUX." If this equals size computed via _osmajor, everything
   is fine. Otherwise, we reset size. */
{
    FP p, q;
    int i;
    /* i=1000: set upper limit on string search in memory */
    for (p=(FP)sys_filetab->f, i=1000; i--, p++; )
        if (IS_AUX(p))
            break;
    if (! i) return 1;
    for (q=p, i=1000; i--, q++; )
        if (IS_CON(q))
            break;
    if (! i) return 1;
    /* size of file structure must equal span from AUX to CON */
    if (size != (q - p))
    {
        puts("size based on _osmajor looks wrong");
        size = q - p;
    }
}

printf("Filename              Size      Attr Handles    Owner\n");
printf("--------              ----      ---- -------    -----\n");

do { /* FOR EACH SFT */

    /* FOR EACH ENTRY IN THIS SFT */
    for (i=sys_filetab->num_files, ff=sys_filetab->f;
         i--;
         ((FP) ff) += size)
    if (ff->num_handles)
    {
        if (_osmajor == 2)
```

```
            {
                // didn't bother with struct for DOS2
                FP ff2 = (FP) ff;
                printf("%.8Fs.",    ff2 + 0x04);
                printf("%.3Fs\t",   ff2 + 0x0c);
                printf("%10lu\t",   *((ULONG far *) (ff2 + 0x13)));
                printf("%04X\t",    ff2[0x02]);
            }
            else
            {
                printf("%.8Fs.",    ff->filename);
                printf("%.3Fs\t",   ff->filename + 8);
                printf("%10lu\t",   ff->fsize);
                printf("%04X\t",    ff->fattr);
                printf("%d\t",      ff->num_handles);
                printf("%04X\t",    ff->owner_psp);
                if (! is_psp(ff->owner_psp))
                    printf("[NOT PSP]");
                if (orphan(ff))
                    printf("[ORPHAN]");
            }
            // FREEUP code can go here
            printf("\n");
        }
        sys_filetab = sys_filetab->next; /* FOLLOWED LINKED LIST... */
    } while (FP_SEG(sys_filetab) &&
             FP_OFF(sys_filetab) != (unsigned) -1); /* ...UNTIL END */
    return 0;
}
```

A fair amount of the source code in FILES.C is devoted to issues surrounding the DOS version number. In addition to checking for an SFT pointer of FFFF:FFFF, probably returned from the DOS box in OS/2, the program also performs a sanity check to see if the size of the DOS file structure really matches the size we've determined from the DOS version number.

The size of the DOS file structure is determined empirically by locating the strings "AUX" and "CON"—the first two files in the SFT—and subtracting their pointers. If the difference is not equal to the size as determined using the DOS version number, the program complains and resets the variable size. It's a good sign we've never seen the program actually display the warning string "size based on _osmajor looks wrong," even though FILES has been tested in DOS 2.x, 3.x, 4.x, and higher.

Filename From Handle Sometimes you need to know the name of a file, and have only its handle available. One important example of this is when you use the DOS redirection facility to redirect stdout to a file rather than to CON, its normal destination. While the stdout handle is always 1, there is no documented way of telling from inside a running program the name of the file to which that (or any other) handle corresponds.

This next program, H2NAME.C, gets that information by combining several undocumented DOS features. It consists primarily of the function h2name(), which, when passed a PSP and a handle, returns a copy of the filename to which that PSP/handle combination corresponds. Note that only the name and extension are reported: getting a complete pathname with subdirectories is left as an exercise for the reader. (Hint: each SFT entry contains a device info word with a possible drive letter; it also contains sector information for the directory.)

While the function h2name() can be clipped out and used in other programs, H2NAME.C also includes a test driver. If H2NAME.EXE is invoked with a PSP number on the command line (you can get the PSP numbers of different processes by running the MEM program from chapter 3), H2NAME enumerates all open files belonging to that process; otherwise, it enumerates all open files belonging to H2NAME itself (using the _psp global found in most C compilers for the PC). To see anything interesting, you often must redirect H2NAME's output to a file, so that this redirected-output file shows up in the enumeration. H2NAME sends its output to stderr, so the output is still visible:

```
C:\UNDOC2>h2name > foo.bar < h2name.c
Files for 726E
 0 ==> H2NAME  C
 1 ==> FOO     BAR
 2 ==> CON
 3 ==> AUX
 4 ==> PRN
 ...
```

H2NAME.C must be compiled with a large memory model so that the memcpy() library routine is capable of dealing with far pointers (alternatively, you could use Microsoft C 6.0 functions such as _fmemcmp(), or loop over the couple of characters yourself):

```
/*************************************************************
 *
 *     H2NAME.C - Jim Kyle
 *
 *  Compile only with large memory model:
 *        tcc -ml h2name
 *        cl -AL h2name.c
 *
 *************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#ifdef __TURBOC__
#include <mem.h>
#else
#include <memory.h>
#endif

#ifndef MK_FP
#define MK_FP(s,o) ((void far *)\
    (((unsigned long)(s) << 16) | (unsigned)(o)))
#endif

char * h2name( unsigned psp, int h )
{ static char name[15];  /* will hold file's name          */
  char far * htbl;
  unsigned far *ptr, nmofs;
  char far *sptr;
  int sftn, sftsize;
  union REGS regs;
  struct SREGS sregs;

  memset( name, 0, 15 ); /* blank out the static name      */

  /* create pointer to handle table (JFT)                   */
  htbl = *((char far * far *) MK_FP(psp, 0x34));

  regs.h.ah = 0x52;      /* set up initial SFT pointer     */
  segread( &sregs );
  intdosx( &regs, &regs, &sregs );
  ptr = *((unsigned far * far *) MK_FP( sregs.es, regs.x.bx + 4 ));

  switch( _osmajor )     /* switch sizes, offsets for ver  */
    { case 2: sftsize = 0x28;
              nmofs = 4; /* offset of 11-byte name area    */
              break;
```

```
      case 3: sftsize = 0x35;
              nmofs = 0x20;   /* offset of name area        */
              break;

      case 4:
      case 5: sftsize = 0x3B;
              nmofs = 0x20;   /* offset of name area        */
              break;

      default: return name;   /* returns null string        */
    }

  if (htbl[h] >= 0)           /* now if handle is valid...  */
    { sftn = htbl[h];         /* get index into SFT list    */
      while ( FP_OFF(ptr) != 0xFFFF )
        { if (ptr[2] > sftn)  /* then target is here        */
            { sptr = (unsigned char far *)&ptr[3];
              while (sftn--)  /* so skip down to it         */
                sptr += sftsize;
              memcpy( name, &sptr[nmofs], 11 );
              return name;    /* found and copied, done     */
            }
          sftn -= ptr[2];     /* not here, reduce index     */
          ptr = (unsigned int far *) MK_FP( ptr[1], ptr[0] );
        }
    }
  strcpy( name, "UNKNOWN" );
  return name;                /* reached only by error      */
}

void main( int argc, char *argv[] )
{
    unsigned psp;
    int max_files;
    int i;

    if (argc < 2)
        psp = _psp;               /* display files for this program */
    else
        sscanf(argv[1], "%X", &psp); /* take PSP from command line */

    if (_osmajor >= 3)
        max_files = *((unsigned far *) MK_FP(psp, 0x32));
    else
        max_files = 20;
```

```
    fprintf(stderr, "Files for %04X\n", psp);

    for (i=0; i<max_files; i++)
        fprintf(stderr, "%2d ==> %s\n", i, h2name(psp, i));
}
```

Here's how h2name() works: after blanking out the static buffer, in order to guarantee a null-string response in case of errors, the function creates a far pointer to the handle table (JFT) for the given process. A pointer to the JFT (and usually the JFT itself) is contained in the PSP. h2name() then sets up the SFT pointer, and the two variables establish SFT size and the position within the SFT of the file or device name, based on the DOS version in use.

With all these preliminaries out of the way, h2name() uses the supplied handle value to index into the handle table, and if the value found there is non-negative (indicating a valid handle), it is used as the SFT index.

The program then walks through the linked list of SFTs until it finds the SFT containing the desired index. Each time it skips over an SFT, the number of entries in the skipped block is subtracted from the desired index, so that the index is always relative to the current block rather than to the absolute beginning of the SFT linked list.

When the correct block is found, a pointer is set to the first byte of its first SFT entry, and then SFT entries are skipped, decrementing the index each time, until the index reaches zero. When this happens the SFT entry under the pointer is the one we're looking for. The name-field offset value is then added to sptr, the 11 bytes at the resulting location are copied into the static buffer "name," and the program returns a pointer to the first byte of the buffer.

Clearly, this same technique could be applied to other information found in the SFT: h2attr(), for example, would return the file attributes rather than the filename.

Making Alterations

All the utilities and functions presented so far in this chapter report back on the state of the DOS file system: we've enumerated all drive letters, determined the values of FILES= and BUFFERS=, enumerated all open files, and turned ordinary file handles into filenames.

Now it's time to *do* something. In this section, another set of utilities is presented, that actually change the file system. One utility bangs on the CDS, one on

the SFT, and one on a JFT. But it's perverse to classify utilities by the data structures they alter: in fact, one utility alters or removes drive mappings, one frees up orphaned file handles, and other can be used to increase a process' handle count.

So, get yourself another cup of coffee and another slice of pizza, and proceed to the next installment in our saga of the DOS file system.

Manufacturing and Removing Drive Letters

Sometimes it is useful to convince MS-DOS that a logical drive is no longer present. If you have ever worked with the Microsoft CD-ROM Extensions (MSCDEX), for example, you might have noted that the only way to deinstall this utility is by rebooting the machine: even TSR management programs like the superb MARK/RELEASE from TurboPower Software are insufficient to remove MSCDEX because, in addition to grabbing memory, MSCDEX also creates a logical drive and that, too, must be undone. Here is an example of DRVOFF in action:

```
C:\UNDOC2>dir d:\

 Volume in drive D is RAMANUJAN
 Directory of  D:\

CHAP1     DOC      4439   6-12-90   9:05a
        1 File(s)   1261568 bytes free

C:\UNDOC2>drvoff d:

C:\UNDOC2>dir d:\
Invalid drive specification
```

All DRVOFF does is call the currdir() function from earlier in this chapter and set the flags word to zero, instantly making the drive invalid. This is the utility that interested one of the authors enough in undocumented DOS to start work on this book (see the Introduction).

At the same time, access to the CDS can be used for more than just invalidating drive letters. With a utility called DRVSET, invalid drives can be activated in an odd way, simply by turning on some bits in the flags word. DOS immediately recognizes the resulting "air drive" as valid, in that you can change to it and send it requests (all of which fail, of course):

```
C:\UNDOC>dir e:
Invalid drive specification

C:\UNDOC>drvset e: netphys
NET PHYSICAL

C:\UNDOC>dir e:

 Volume in drive E has no label
 Directory of  E:\

File not found

C:\UNDOC>e:

E:\>chkdsk
Cannot CHKDSK a Network drive
```

Simply by twiddling bits in the CDS, we convince DOS that E: is in some way now a valid drive. But since DIR doesn't show any files up there, it's not clear what value the drive has. In fact, this is the foundation for creating drives with the network redirector. Simply by setting drive E: to be a "network" drive (again, the term "network" really just means a non-FAT file system), we can route all DOS file requests for the drive to an INT 2Fh Function 11h handler. Later on in this chapter, we will write such a handler. In the meantime, DRVSET is crucial for experimenting with air drives. Such "air drives" are the foundation for installable file systems.

Since DRVOFF and DRVSET are so similar, it makes sense to package them in the same source module. Compile DRVSET.C below with Turbo C or Microsoft C, link with CURRDIR.OBJ from earlier in the chapter, and copy the resulting DRVSET.EXE to DRVOFF.EXE. The resulting program responds differently depending on whether its name (argv[0] in C) is DRVSET or DRVOFF:

```
/* DRVSET.C -- set attrib of drive given on command line */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "currdir.h"

void fail(char *s) { puts(s); exit(1); }
```

```
main(int argc, char *argv[])
{
    CDS far *drv;
    unsigned drive;
    int drvoff;

    /* to just turn off drives, program can be renamed DRVOFF */
    drvoff = strstr(strupr(argv[0]), "DRVOFF");

    /* what drive do they want? (accepts letters and numbers) */
    if (argc < 2)
    {
        if (drvoff)
            fail("usage: drvoff [drive]");
        else
            fail("usage: drvset [drive] <NET|PHYS|SUBST|JOIN|OFF>");
    }
    else if (argv[1][0] >= 'A')
        drive = toupper(argv[1][0]) - 'A';
    else
        drive = atoi(argv[1]);

    if (! (drv = currdir(drive)))
        fail("can't get current directory structure");

    /* just turn drive off */
    if (drvoff)
    {
        drv->flags = 0;
        return 0;
    }

    /* change drive state */
    if (argc > 2)
    {
        strupr(argv[2]);
        if (strstr(argv[2], "OFF"))     drv->flags = 0;
        if (strstr(argv[2], "NET"))     drv->flags |= NETWORK;
        if (strstr(argv[2], "SUBST"))   drv->flags |= SUBST;
        if (strstr(argv[2], "JOIN"))    drv->flags |= JOIN;
        if (strstr(argv[2], "PHYS"))    drv->flags |= PHYSICAL;
    }

    /* print current drive state */
    if (! drv->flags)           fputs("INVALID ", stdout);
    if (drv->flags & NETWORK)   fputs("NET ", stdout);
    if (drv->flags & SUBST)     fputs("SUBST ", stdout);
```

```
    if (drv->flags & JOIN)       fputs("JOIN ", stdout);
    if (drv->flags & PHYSICAL)   fputs("PHYSICAL ", stdout);
    putchar('\n');

    return 0;
}
```

Releasing Orphaned File Handles

Because DOS does not automatically close all files that belong to a process when that process terminates and stays resident, the common practice of redirecting output from a TSR's installation code to the NUL device normally "loses" one handle in the SFT. Those handles, known as *orphans*, can be made available once again using a modified version of the FILES program presented earlier in this chapter. The modification is quite small, and goes just before the line that reads printf("\n"):

```
#ifdef FREEUP
    // only DOS 3+
    if (! IS_AUX(ff->filename))
    if (! IS_CON(ff->filename))
    if (! IS_PRN(ff->filename))
    if (orphan(ff) || (! is_psp(ff->owner_psp)))
    {
        if (! (-- ff->num_handles))  // decrement owners
            printf(" [FREED]");
        else
            printf(" [NOW %d]", ff->num_handles);
    }
#endif
```

To produce FREEUP.EXE, add the above code to FILES.C and compile with the following command lines:

Microsoft C:

```
cl -DFREEUP -Fefreeup.exe files.c
```

Turbo C:

```
tcc -DFREEUP -efreeup.exe files.c
```

Assuming that an orphaned NUL handle is still lurking about the SFT, running FREEUP produces the following results. Notice that FREEUP doesn't do anything stupid like free up the file for its redirected output, and that AUX,

CON, and PRN don't get changed, even though they have invalid owner PSPs and are therefore otherwise perfect candidates for being freed:

```
C:\UNDOC>freeup > freeup.log
C:\UNDOC>type freeup.log
Filename              Size      Attr Handles    Owner
--------              ----      ---- -------    -----
AUX     .                0      0000    6       9DED    [NOT PSP]
CON     .                0      0000    16      9DED    [NOT PSP]
PRN     .                0      0000    6       9DED    [NOT PSP]
NUL     .                0      0000    1       0AE9    [ORPHAN] [FREED]
FREEUP  .LOG             0      0020    2       0AE9

C:\UNDOC>files
Filename              Size      Attr Handles    Owner
--------              ----      ---- -------    -----
AUX                      0      0000    6       9DED    [NOT PSP]
CON                      0      0000    16      9DED    [NOT PSP]
PRN                      0      0000    6       9DED    [NOT PSP]
```

FREEUP can be quite useful in AUTOEXEC.BAT files where you want to discard the TSR's initialization output without losing file handles:

```
tsr > nul
freeup > nul
```

However, note that FREEUP is not necessary with the TSRs produced using the TSR skeleton in chapter 5. As noted in that chapter, the acid test for correct TSR deinstallation is the freeing up of any otherwise orphaned file handles. The generic TSR in chapter 5 deinstalls using a normal DOS terminate (INT 21h Function 4Ch), thereby closing and freeing any open file handles.

More File Handles

We've seen that the handle-based file I/O routines introduced in DOS 2.0 rely on the existence of two data structures: the system-wide linked list of System File Tables (SFTs) and the table of file handles (Job File Table, JFT) owned by each process (PSP). In contrast to the older File Control Blocks (FCBs), which allocated on an as-need basis by applications, the SFTs and each JFT are normally allocated by DOS itself, and therefore are limited in size. We've seen that the number of files held in the SFTs is controlled by the FILES= statement in CONFIG.SYS. The num-

ber of possible open file handles in a process' JFT is normally 20. This number is dictated by the fact that the array resides directly in the process' PSP—this is the downside to the switch from FCBs to handles/SFTs.

DOS 3.3 introduced a function, Set Handle Count (INT 21h Function 67h), which can increase the size of the calling process' JFT, thereby increasing the number of files and devices that may by open simultaneously using handle-based file I/O. Sometimes, programmers think the function doesn't work merely because they forgot to increase the FILES= setting before attempting to keep 50 files open at once. However, as indicated in Appendix A, there really are bugs in this function that often preclude its use. A *PC Tech Journal* article from the time (April 1988) noted that the function can incorrectly allocate 64KB too much memory because its code uses an ROR instruction instead of the correct RCR.

Fortunately, it is easy to perform the same function yourself. True, the open file table is embedded directly in the PSP, so it seems it would be difficult to increase its size. However, since DOS 3.0, the PSP has also contained a far pointer to the open file table, and a word holding its size. The relevant fields in the PSP are:

```
18h 20 BYTEs    DOS 2+ open file table (JFT), FFh = unused
32h    WORD     DOS 3+ max open files
34h    DWORD    DOS 3+ open file table (JFT) address
```

You can't do anything to increase the size of the array at offset 18h in the PSP, but you can allocate a new, larger block of memory for the JFT, bump up the count at offset 32h in PSP, copy the old table into the new one, and then set the pointer at offset 34h to the new table. (The same type of manipulation is possible with other seemingly static DOS arrays, such as the SFTs and CDS, whose far pointers are located in the List of Lists.) This series of operations is carried out by the following program, FHANDLE.C:

```
/*
FHANDLE.C
Alternative to using INT 21h Function 67h (added in DOS 3.3)
*/

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

typedef unsigned char BYTE;
```

```
typedef unsigned WORD;
typedef unsigned long DWORD;
typedef BYTE far *FP;

#ifndef MK_FP
#define MK_FP(seg,ofs)  ((FP)(((DWORD)(seg) << 16) | (ofs)))
#endif

extern unsigned files(void);    // in COUNTF.C

void fail(char *s) { puts(s); exit(1); }

main(int argc, char *argv[])
{
    int f;
    int i;

    BYTE far *tbl = MK_FP(_psp, 0x18);
    WORD far *pmax = MK_FP(_psp, 0x32);
    BYTE far * far *ptbl = MK_FP(_psp, 0x34);
    BYTE far *fp, far *p;
    WORD max = *pmax;
    WORD new_max = atoi(argv[1]);

    printf("Currently %u max file handles\n", max);

    if (new_max <= max)
        fail("nothing to do");

    // make sure proposed JFT size is <= SFT size
    if (new_max > files())
        fail("FILES= too low: edit CONFIG.SYS and reboot");

    if (! (fp = (BYTE far *) malloc(new_max)))
        fail("insufficient memory");
    if (tbl != *ptbl)
        tbl = *ptbl;
    for (i=0, p=fp; i<max; i++, p++)
        *p = tbl[i];
    for ( ; i < new_max; i++, p++)
        *p = 0xFF;
    *pmax = new_max;
    *ptbl = fp;

    printf("Max file handles increased to %u\n", new_max);

    // now test how many files we can open
```

```
    for (i=0; ; i++)
        if (_dos_open(argv[0], 0, &f) != 0)
            break;
    printf("Opened %d files\n", --i);

#ifdef TESTING
    _dos_close(f);  // close last one so we can spawn shell!
    system(getenv("COMSPEC"));
#endif

    return 0;
}
```

To test that more files can actually be opened, FHANDLE continually opens its own executable file (argv[0]) in a loop until the Microsoft C _dos_open() function fails. This is how FHANDLE behaves on a system with FILES=40 in CONFIG.SYS:

```
C:\UNDOC>fhandle 40
Currently 20 max file handles
Max file handles increased to 40
Opened 34 files
```

FHANDLE also uses the files() function from COUNTF (found earlier in this chapter) to make sure that it makes sense to increase the size of the JFT. It is important to note that an enlarged JFT is not inheritable, so FHANDLE can't pass its increased wealth along to any children it might have.

Indirect Server Call

The next program, written by Ralf Brown, shows how two undocumented calls can be put together to fill a gap in the DOS programmer's interface: renaming or moving groups of files. DOS has two different functions for renaming files, but they have complementary limitations: on the one hand, the earlier Rename File function (INT 21h Function 17h) accepts file wildcards, and thus can rename entire groups of files with a single call, but it is based on FCBs and therefore can't access a directory structure. On the other hand, the later Rename File function (INT 21h Function 56h) knows all about directory structures, but can only handle one file at a time. Isn't there a simple way to rename (move) an entire group of files across subdirectories without looping over each file?

Curiously, there is. As described in the appendix, the later Rename File function, when invoked via INT 21h Function 5Dh Subfunction 00h (Server Function Call), has the undocumented behavior of allowing wildcards in both the source and destination file specifications. INT 21h Function 41h (Delete File) has the same undocumented behavior as well.

INT 21h Function 5Dh Subfunction 00h (Server Function Call) executes a specified INT 21h call for a specified network machine number and process ID, so at first it sounds as if this function is useful only in networked DOS environments. However, machine number 0 specifies the current machine, so this function can be used locally as well, or even in a stand-alone DOS environment without a network (are there really any non-networked machines left?):

```
/*
MOV.C
demonstrate wildcard pathed renames via DOS indirect
function call
by Ralf Brown, with thanks to Dan Lanciani for pointing out that
indirect function call enables wildcards on rename and delete

Usage: MOV old-filespec new-filespec
*/

#include <stdio.h>
#include <dos.h>

typedef struct
{
   unsigned ax,bx,cx,dx,si,di,ds,es,reserved,computerID,processID ;
} DPL ;     /* DOS parameter list */

union REGS regs ;
struct SREGS segregs ;

void canonicalize(filespec,canonical,errorlevel)
char *filespec, *canonical ;
int errorlevel ;
{
   regs.h.ah = 0x60 ;
   regs.x.si = FP_OFF((void far *)filespec) ;
   segregs.ds = FP_SEG((void far *)filespec) ;
   regs.x.di = FP_OFF((void far *)canonical) ;
   segregs.es = FP_SEG((void far *)canonical) ;
   intdosx(&regs,&regs,&segregs) ;
```

```
   if (regs.x.cflag)
   {
      puts("invalid filespec") ;
      exit(errorlevel) ;
   }
}

#define ERROR(s, x)     { puts(s); errorlevel = (x); }

int main(argc,argv)
int argc ;
char **argv ;
{
   DPL dpl ;
   void far *ptr ;
   int errorlevel = 0 ;
   char source[128], target[128] ;

   if (argc != 3)
   {
      puts("usage: MOV old-filespec new-filespec") ;
      puts("where old-filespec and new-filespec may contain") ;
      puts("wildcards. Wildcards in the new-filespec indicate") ;
      puts("that new name should contain same characters as") ;
      puts("old name in those positions.") ;
      errorlevel = 1 ;
   }
   else if (_osmajor < 3 || (_osmajor == 3 && _osminor < 10))
      ERROR("MOV requires DOS 3.10 or higher", 5) ;
   else
   {
      canonicalize(argv[1],source,3) ;
      canonicalize(argv[2],target,4) ;
      if (source[0] != target[0])    /* are they on the same drive? */
        ERROR("Source and target must be on the same drive", 6) ;
      else /* do the move/rename */
      {
        dpl.ax = 0x5600 ;   /* indirect function is rename */
        dpl.dx = FP_OFF((void far *)&source) ;
        dpl.ds = FP_SEG((void far *)&source) ; /* DS:DX old filespec */
        dpl.di = FP_OFF((void far *)&target) ;
        dpl.es = FP_SEG((void far *)&target) ; /* ES:DI new filespec */
        dpl.bx = dpl.cx = dpl.si = 0 ;
        dpl.computerID = 0 ;    /* local machine */
        dpl.processID = 0 ;     /* current process */

        regs.x.ax = 0x5D00 ;  /* invoke server function call */
```

```
        ptr = (void far *)&dpl ;
        regs.x.dx = FP_OFF(ptr) ;
        segregs.ds = FP_SEG(ptr) ;
        intdosx(&regs,&regs,&segregs) ;
        /* rename returns error 12h (no more files) on success */
        if (regs.x.cflag && regs.x.ax != 0x12)
            ERROR("rename failed", 2) ;
        /* NOTE: fails in OS/2 DOS box */
      }
   }
   return errorlevel ;
}
```

This example shows that whereas a direct DOS call is performed by moving values into the CPU registers, an indirect DOS call is performed by filling in the contents of the DOS parameter list (DPL) with the values that you would otherwise move into the CPU registers. To indirectly invoke INT 21h Function 56h, instead of setting AX to 5600h, we set dpl.ax to 5600h. DS:DX is then set to the address of the DPL and we invoke INT 21h with AX=5D00h. Note the similarity between this extra level of indirection and the indirect call blocks used in chapter 2 in the section on "Undocumented DOS calls from Protected Mode." Of course, the C union REGS structure itself is just another example of an indirect DOS call block, in which one changes an image of the CPU registers rather than the registers themselves.

The MS-DOS Network Redirector

Finally, we turn to the MS-DOS network redirector. Not to be confused with "redirection," a *redirector* is a mechanism for inserting a monitor into the stream of file system requests, so that some requests can be pulled out and serviced in a special way. Generally, such requests are serviced by being transformed into network packets and sent to a file server, but there is really nothing in the process of redirection which restricts it to networking.

The redirector interface has been used by Microsoft since DOS 3.1 to allow alien file systems to be transparently accessible by DOS programs. The fact that this is concurrent with the version of DOS that first supported networking is no coincidence, since the redirector interface is the vehicle DOS provides for the implementation of network services such as IBM PC LAN.

Since that time, the High Sierra and ISO 9660 CD-ROM directory and file formats, which support >600MB files and volumes, have been implemented using

the redirector interface, in the Microsoft CD-ROM Extensions program (MSCDEX.EXE). Other non-Microsoft users of the interface include Banyan and 3Com in their respective network operating systems.

It is worth stressing at this juncture the usefulness and power of the redirector interface. Although arcane, inconsistent, and just plain awkward in places as we will see, the possibilities for its application are enormous. So why is it not widely used? The answer is that Microsoft has not documented it, and will not support it. It is even claimed by Microsoft insiders that it is not documented internally, except through hearsay and "oral history." Evidently, then, the information and techniques presented in this chapter are empirically derived, incomplete in places, and will no doubt have to be updated as additions and revisions to our knowledge about the interface become known. The primary tools used were Ralf Brown's listings for Appendix A of this book, and the INTRSPY program in chapter 8.

Because of the possibilities for confusion in terminology, let us define some terms from the outset. From here on in, the term "redirector interface" means the functionality and hooks provided by DOS, and "a redirector" or "the redirector" means any given program that uses the interface. Concretely, DOS will at certain times call INT 2Fh Function 11h: this is called the *redirector interface*. Programs can receive these calls from DOS by taking over INT 2Fh Function 11h: these are called *redirectors*.

After the following brief word on the subject, the term Installable File System (IFS) will not be used again, in order to keep the terminology to manageable levels. IFS is a term that was introduced with OS/2 1.2, and allows specialized file systems to be developed and to link in seamlessly. The first file system to use OS/2 IFS is the High Performance File System (HPFS), designed as a replacement for the old FAT file system. In essence, IFS was to be a legitimized version under OS/2 of the hidden, undocumented redirector interface under DOS. The fact that the initials IFS appear frequently in DOS 4.0 documentation indicates that the intention was there to provide some form of stepping stone to help developers migrate towards OS/2 (particularly LAN Manager).

The IFS interface appears to be implemented under DOS 4 via the IFSFUNC program, which loads itself as a redirector. Some additional subfunctions have appeared under the redirector interface to support what may be enhanced functionality over and above the existing redirector interface. However, IFS in the DOS world is receiving no publicity at all, and it is probably fair to say that DOS

4.0 is being retired early due to lack of enthusiasm. There is not much benefit to a discussion of IFSFUNC, REDIRIFS, or any other DOS 4.x-specific spin put on the network redirector. It is sensible to consider the IFS interface only in so far as it is visible, and in that it overlaps the redirector interface. In other words, subfunctions of Int 2Fh function 11h will be referred to as redirector interface subfunctions, even if they were first introduced in DOS 4.0 as IFS subfunctions.

What is the Redirector Interface, and How Do We Use It?

Basically, we want to use the redirector interface to manufacture DOS drives. A DOS drive is any entity that has a drive letter, a CDS entry, and which behaves like hard drives or floppies, in that we can perform normal DOS disk, directory, and file operations. Whether there is actually magnetic media at the other end is irrelevant.

Almost any area in computing can be considered as a set of file operations. To take one example, let's say that every evening you log onto an information service such as CompuServe or BIX. These services have a hierarchical structure of forums, message and library areas within forums, topics within message areas, and so on. Normally to access such services, you dial up via modem and send the service various command strings or menu selections. However, it might be more convenient to pretend that the information service is just another drive on your machine. When you typed CD \IBM.DOS\SECRETS, for example, it might be equivalent to joining the ibm.dos/secrets conference on BIX.

Conceptually, this is quite simple: all we need is to designate a drive letter for the information service (I:, for instance) and install a piece of software that will catch all disk, directory, and file requests sent to that drive. Remember that DOS merely defines this specification; any given redirector must supply the actual functions that meet this specification.

Some redirectors are implemented in precisely this way: by hooking INT 21h and watching every function call that comes in. Novell NetWare works this way. However, there are problems with simply hooking INT 21h and looking for all file-related calls for your special drive. For instance, you must separately deal with FCB- and handle-based calls. You also must maintain a fairly large amount of state, just for correctly handling complex path names.

Another way to attach foreign file systems to DOS is of course to use an installable block device driver. But this imposes numerous restrictions on the file system: it has to have a BPB, DPB, FAT, all sorts of other pieces that are simply

not appropriate to all file systems. There is the added fact that device drivers are just plain inconvenient (though DEVLOD helps a lot!).

The alternative to hooking INT 21h or writing a device driver is to use the redirector interface. Providing fictional drive mappings is what the redirector interface is all about.

On one side of the redirector interface are the redirector services. These consist of DOS data structures and a set of function calls. The usual model of an operating system is that of a lower-level program that responds to requests for services (function calls in DOS parlance) initiated by an application. The redirector interface, however, specifies function calls that are generated by the operating system—calls that a redirector may intercept and service. These calls are implemented under the multiplex interrupt, INT 2Fh, and not, as commonly supposed, under INT 21h. Function 11h of the multiplex interrupt is set aside for redirector services, and each redirector function call is a subfunction of Function 11h.

The data structures include the List of Lists (LoL), Current Directory Structure (CDS), the System File Table (SFT), and the Swappable Data Area (SDA).

As we know, the DOS List of Lists structure, obtained through undocumented INT 21h Function 52h, provides the address of the CDS table and the LASTDRIVE value. A redirector is responsible for initializing, maintaining, and, if it is de-installable, restoring the CDS for its chosen drive letter(s). You can only redirect a drive that has an entry in the CDS table. DOS is primarily interested in three fields in the CDS: the current directory string, the offset of the root directory in that string, and the flags word.

An SFT entry holds the state that DOS maintains for each open file in the system. DOS again appears to only be interested in a subset of fields in the structure, in particular the open-mode flags (which describe the access level to, and shareability of, the file), the device information flags word (which indicates whether the device is a block or character level device, whether it has been written to, etc.), and the date, time, and file size fields.

Note that there are fields in the above structures that appear to be wholly for the use by the drive's owner. Specifically, they appear to be designed for use by DOS in managing DOS file format drives, and deal with units of sector and cluster. If the redirector device is not formatted or structured in that way, those areas of the structures can apparently be used in whatever way the redirector chooses.

The final, most important structure in the interface is the SDA. DOS is usually non-reentrant, but it can be re-entered using the SDA (this is explained in

more detail in chapter 5, which includes a section on using the SDA to build TSRs). The SDA is that area of the DOS data segment which must be saved and restored to provide for DOS reentrancy. More than that, however, it is also the part of the DOS data segment which contains all the global data required for implementation of a redirector. That part amounts to only some 10 to 15 fields in the structure.

Here is an INTRSPY script file that describes the structures that are involved, and that will be included in other script files used to investigate the interface:

```
;; Current Directory Structure entry - All DOS versions
;; also see Appendix A entry for INT 21h Function 5D06h, 5D0Bh
structure CDS fields
        CURR_PATH (byte,asciiz,67)
        FLAGS (word,bin)
        DISK_BLK (dword)
        INFO_PTR (dword,ptr)
        f1 (word)
        ROOT_OFS (word,dec)
; In DOS 4.0 and above there are a further 7 bytes of
; IFS/SHARE fields

;; System File Table entry - All DOS versions
structure SFT fields
    C_HANDLES (word,dec)
    OPEN_MODE (word)
    ATTR_BYTE (byte)
    DEV_INFO (word)
    DPB_PTR (dword,ptr)
    ST_CLSTR (word,dec)
    F_TIME (word)
    F_DATE (word)
    F_SIZE (dword,dec)
    F_POS (dword,dec)
    LAST_RELCLSTR (word,dec)
    LAST_ABSCLSTR (word,dec)
    DIR_SCTR_NO (word,dec)
    DIR_ENTRY_NO (byte,dec)
    FCB_FNAME (byte,ascii,11)

;; Swappable DOS Area - DOS versions 3.1 to 3.3 - Used
;;  in the form SDA_SEG:SDA_OFS->SDA3
structure SDA3 fields
         CRITERR_FLAG (byte,dec)
         INDOS_FLAG (byte,dec)
         DRIVE_NO (byte,dec)
         LASTERR_DUM (byte,,9)
         CURR_DTA   (dword,ptr)
         CURR_PSP   (word)
         SP_X_INT23 (word)
```

```
LAST_RC (word)
CURR_DRIVE (byte)
EXTBRK (byte)
INT21_AX (word)
SHRNET_PSP (word)
NET_MC_NO (word)
MEM_BLK_DATA (word,,3)
DONTKNOW1 (byte,,10)
DD (byte,dec)
MM (byte,dec)
YY_1980 (word,dec)
D_1_1_1980 (word,dec)
D_O_W (byte,dec)
DONTKNOW2 (byte,,3)
DEV_REQ_H1 (byte,dump,26)
DEV_DRVPTR (dword,ptr)
DEV_REQ_H2 (byte,dump,22)
DEV_REQ_H3 (byte,dump,22)
DONTKNOW3 (byte,,8)
CLOCK_TXREC (byte,,6)
DONTKNOW3A (byte,,2)
FN1 (byte,asciiz,128)
FN2 (byte,asciiz,128)
SDB (byte,dump,21)
FOUND_FILE (byte,dump,32)
DRIVE_CDSCOPY (byte,dump,81)
FCB_FN1 (byte,ascii,11)
DONTKNOW4 (byte)
FCB_FN2 (byte,ascii,11)
DONTKNOW5 (byte,,11)
SRCH_ATTR (byte)
OPEN_MODE (byte)
DONTKNOW6 (byte,,3)
CALL_TYPE (byte)
DONTKNOW7 (byte,,9)
TERM_PROCTYP (byte)
DONTKNOW8 (byte,,2)
CRITERR_DPBPTR (dword,ptr)
INT21_SS_SP (dword,ptr)
DONTKNOW9 (byte,,14)
MEDIA_ID (byte)
DONTKNOW10 (byte)
CURR_SFTPTR (dword,ptr)
DRIVE_CDSPTR (dword,ptr)
DONTKNOW11 (byte,,8)
JFT_PTR (dword,ptr)
FN1_CSOFS (word,hex)
FN2_CSOFS (word,hex)
DONTKNOW12 (byte,,46)
BX_DS_TMP (word,,3)
PREV_STACK (dword,ptr)
REN_SRCFILE (byte,,21)
REN_FILE (byte,,32)
```

```
;; Swappable DOS Area - DOS versions 4.0 onwards - Used
;;  in the form SDA_SEG:SDA_OFS->SDA4
structure SDA4 fields
        CRITERR_FLAG (byte,dec)
        INDOS_FLAG (byte,dec)
        DRIVE_NO (byte,dec)
        LASTERR_DUM (byte,,9)
        CURR_DTA   (dword,ptr)
        CURR_PSP   (word)
        SP_X_INT23 (word)
        LAST_RC (word)
        CURR_DRIVE (byte)
        EXTBRK (byte)
        DONTKNOW0 (word)
        INT21_AX (word)
        SHRNET_PSP (word)
        NET_MC_NO (word)
        MEM_BLK_DATA (word,,3)
        DONTKNOW1 (byte,,10)
        DD (byte,dec)
        MM (byte,dec)
        YY_1980 (word,dec)
        D_1_1_1980 (word,dec)
        D_O_W (byte,dec)
        DONTKNOW2 (byte,,3)
        DEV_REQ_H1 (byte,dump,30)
        DEV_DRVPTR (dword,ptr)
        DEV_REQ_H2 (byte,dump,22)
        DEV_REQ_H3 (byte,dump,30)
        DONTKNOW3 (byte,,6)
        CLOCK_TXREC (byte,,6)
        DONTKNOW3A (byte,,2)
        FN1 (byte,asciiz,128)
        FN2 (byte,asciiz,128)
        SDB (byte,dump,21)
        FOUND_FILE (byte,dump,32)
        DRIVE_CDSCOPY (byte,dump,88)
        FCB_FN1 (byte,ascii,11)
        DONTKNOW4 (byte)
        FCB_FN2 (byte,ascii,11)
        DONTKNOW5 (byte,,11)
        SRCH_ATTR (byte)
        OPEN_ATTR (byte)
        DONTKNOW6 (byte,,3)
        CALL_TYPE (byte)
        DONTKNOW7 (byte,,9)
        TERM_PROCTYP (byte)
        DONTKNOW8 (byte,,3)
        CRITERR_DPBPTR (dword,ptr)
        INT21_SS_SP (dword,ptr)
        DONTKNOW9 (byte,,16)
        MEDIA_ID (byte)
        DONTKNOW10 (byte,,5)
```

```
CURR_SFTPTR (dword,ptr)
DRIVE_CDSPTR (dword,ptr)
DONTKNOW11 (byte,,8)
JFT_PTR (dword,ptr)
FN1_CSOFS (word,hex)
FN2_CSOFS (word,hex)
DONTKNOW12 (byte,,52)
BX_DS_TMP (word,,3)
PREV_STACK (dword,ptr)
DONTKNOW13 (byte,,9)
SPOP_ACT (word)
SPOP_ATTR (word)
SPOP_MODE (word)
DONTKNOW14 (byte,,29)
REN_SRCFILE (byte,,21)
REN_FILE (byte,,32)
```

On the other side of the interface is the redirector itself, be it MSCDEX, the PC LAN program, the hypothetical CompuServe file system, or this chapter's demonstration redirector, the Phantom. A redirector will normally load itself as a TSR program, and install itself in the chain of INT 2Fh handlers. It will, in other words, get the vector to the current INT 2Fh handler, store it as the next handler in the chain, and then set the INT 2Fh vector to point to itself. When INT 2Fh is invoked, it will receive control. If the call is not for Function 11h (AH=11h), control will be passed to the next handler in the chain. In this way, the redirector will monitor all INT 2Fh calls, and will filter out all but redirector interface function calls.

The redirector interface subfunctions are dealt with individually a little later in the chapter, but an initial discussion of their characteristics is appropriate at this point. DOS file functions available to applications provide two very different interfaces. Use of the older more traditional FCB style call has the following implications:

- (practically) limitless concurrent open files because the state for the open file is kept in a user supplied structure
- wildcard filename specification in the Delete and Rename FCB functions
- filenames cannot contain pathnames

Use of file handle calls implies:

- concurrent open file count is limited to the FILES= line in CONFIG.SYS (that is, to the size of the SFT), and, for any given applications, to the size of its JFT

- filenames can contain pathnames
- simplicity of use ("magic cookie")
- DOS I/O redirection and piping

The redirector interface unifies the two access methods so that a redirector need not know by what method a file is being accessed. This information helps us to visualize at what level the interface is functioning, that is, below the user interface, at the file access method level, and above the physical device layer. This access method independence is a great labor saver, and confirms the desirability of the redirector interface over INT 21h replacement as the means of implementing alternative file systems. Rather than having to duplicate the entire range of function calls in the DOS programmer's interface, which includes FCB- and handle-based file access methods, a redirector plugs in at a level where much of the higher level administrative functionality has been stripped away, and only a stream level interface need be dealt with. This means that all filename strings used to open and otherwise manipulate files contain fully qualified paths at the redirector interface; the work of resolving drive and directory has already been done by the DOS kernel.

It is often stated that the network redirector "grabs file system calls for the remote drive before INT 21h sees them," or words to that effect. Actually, the redirector operates at a level *below* INT 21h. The code for INT 21h takes care of calling INT 2Fh Function 11h when appropriate.

Tracing an Open

In order to see the interface in action, you can use an INTRSPY script to follow a File Open call. Ensure that a redirector program such as MSCDEX or PC LAN is loaded and that you can list files on the redirected device, that is, the CDROM drive or the remote server. Then, using as a parameter the name of a file that exists on the redirected drive, run the following INTRSPY script (see chapter 8 for detailed instructions on using INTRSPY):

```
;; Intended for use with versions of DOS >=3.1 and <4.00

include "dosstruc"

intercept 21h
    function 3dh
        on_entry
```

```
            output "== DOS OPEN (3Dh) ====================="
            output "File name: " (ds:dx->byte,asciiz,40)
            output "Open mode: " al
        on_exit
            output ""
            output "== 3D OPEN Completed "
            if (cflag == 1) sameline "(FAILED " ax ") ==========="
            if (cflag == 0) sameline "(Handle " ax ") ==========="

intercept 2fh
    function 11h
        subfunction 16h
            on_entry
                output ""
                output "-- 2F Open (16h) -----------------"
                output "File name: " (sda_seg:sda_ofs->SDA3.FN1)
                output "Open mode: " (sda_seg:sda_ofs->SDA3.OPEN_MODE)
                output "Uninitialized SFT:"
                output (es:di->SFT)
            on_exit
                output ""
                output "-- 2F Open completed "
                if (cflag == 0)
                    sameline "-----------------"
                    output "Completed SFT:"
                    output (es:di->SFT)
                if (cflag == 1) sameline "(FAILED " ax ") ---------"

run "command /c type %1"

report "2fopen.out"
```

This will type out the file entered as the parameter. If the file is on a redirected drive, as in the MSCDEX-based example L:\READ.ME, the file 2FOPEN.OUT will contain something like:

```
== DOS OPEN (3Dh) ====================
File name:  d:read.me
Open mode: 00

-- 2F Open (16h) -----------------
File name:  \\D.A.\READ.ME
Open mode:  00h
Uninitialized SFT:
SFT.C_HANDLES            : 65535
SFT.OPEN_MODE            : 0000h
```

```
SFT.ATTR_BYTE              : 20h
SFT.DEV_INFO               : 0042h
SFT.DPB_PTR                : 028E:7620
SFT.ST_CLSTR               : 9933
SFT.F_TIME                 : 6000h
SFT.F_DATE                 : 0F30h
SFT.F_SIZE                 : 25332
SFT.F_POS                  : 25332
SFT.LAST_RELCLST           : 12
SFT.LAST_ABSCLST           : 9945
SFT.DIR_SCTR_NO            : 165
SFT.DIR_ENTRY_NO           : 2
SFT.FCB_FNAME              : COMMAND COM

-- 2F Open completed ----------------
Completed SFT:
SFT.C_HANDLES              : 65535
SFT.OPEN_MODE              : 0002h
SFT.ATTR_BYTE              : 01h
SFT.DEV_INFO               : 8043h
SFT.DPB_PTR                : 0D57:00F3
SFT.ST_CLSTR               : 9933
SFT.F_TIME                 : 8A53h
SFT.F_DATE                 : 1292h
SFT.F_SIZE                 : 21900
SFT.F_POS                  : 0
SFT.LAST_RELCLST           : 21560
SFT.LAST_ABSCLST           : 0
SFT.DIR_SCTR_NO            : 56026
SFT.DIR_ENTRY_NO           : 2
SFT.FCB_FNAME              : READ    ME

== 3D OPEN Completed (Handle 0005) ==========
```

This shows the DOS Open function being called with the raw filename string as specified, and the resultant redirector Open call Subfunction 16h being called with the SDA.FN1 field now reflecting a fully qualified filename (\\D.A\READ.ME above). It also shows an uninitialized SFT being passed to the redirector Open function, with data left over from previous use (COMMAND.COM in this case), and being completed by MSCDEX. Thus, you see what details DOS takes care of before calling a redirector, and what tasks the redirector is responsible for.

Differences Between DOS Versions

The next example works with DOS versions from 3.10, when the redirector interface was introduced, through to DOS 4.0 and higher. The interface has not changed much in that time, except in one or two important areas. Perhaps the most predictable change is that some subfunctions have been added to cater to new functions added into the DOS function interface. In DOS 4.0, the Extended Open (6Ch) function was introduced to allow all types of file open to be available through one call. At that time, a corresponding new redirector interface Subfunction number (2Eh) was added to handle the extended open. The SDA also changed with the introduction of DOS 4.0. While we only know the purpose of some of the fields in the SDA, we can see, following the same example, that the SPECOPEN_ACTION, SPECOPEN_MODE, and SPECOPEN_ATTR fields have been added to support the special open functionality.

Another Subfunction number introduced with DOS 4.0 is 2Dh. It is not at all clear what this function is for, but some of the DOS internal commands use the (also undocumented) DOS Function 57h which appears to trigger 2Dh at the redirector interface. However, both DOS Function 57h and the redirector interface Subfunction 2Dh are amongst those DOS 4.0 calls that disappear in later DOS versions.

Redirector Subfunctions

The following table presents the redirector subfunctions we know about, with usage, parameters, and notes. Remember that DOS merely defines this specification; any given redirector must supply the actual functions that meet this specification:

Subfunction 01h
Remove Directory
Inputs: SDA.FN1 = fully qualified directory name
Outputs: Carry set + error code in AX if error encountered

Subfunction 03h
Make Directory
Inputs: SDA.FN1 = fully qualified directory name
Outputs: Carry set + error code in AX if error encountered

Subfunction 05h
Change Current Directory
Inputs: SDA.FN1 = fully qualified directory name
Outputs: Carry set + error code in AX if error encountered
Note: This function is expected to update the CURR_PATH field of the CDS for the drive. It is advisable to maintain the DOS norm of not terminating the directory string with a '\' except when the current directory is root.

Subfunction 06h
Close File
Inputs: ES:DI -> SFT for file to close
Outputs: Carry set + error code in AX if error encountered
SFT completed if no error
Note: Do not update the C_HANDLES field in the SFT. DOS maintains this field itself.

Subfunction 07h
Commit File
Inputs: ES:DI -> SFT for file to commit (flush buffers)
Outputs: Carry set + error code in AX if error encountered

Subfunction 08h
Read from File
Inputs: ES:DI -> SFT for file to read from
CX = count of bytes to read
SDA.CURR_DTA -> user buffer to read data into
Outputs: Carry set + error code in AX if error encountered
if no error, CX = bytes actually read
SFT updated

Subfunction 09h
Write to File
Inputs: ES:DI -> SFT for file to write to
CX = count of bytes to write
SDA.CURR_DTA -> user buffer to write data from

Outputs: Carry set + error code in AX if error encountered
if no error, CX = bytes actually written
SFT updated

Subfunction 0Ah
Lock Region of File
Inputs: ES:DI -> SFT for file
CX:DX = Starting offset of region
SI = High word of size of region
Word at top of stack = Low word of size of region
BX = file handle
Outputs: Carry set + error code in AX if error encountered
Note: The redirector is expected to perform the task of resolving lock conflicts.

Subfunction 0Bh
Unlock Region of File
Inputs: ES:DI -> SFT for file
CX:DX = Starting offset of region
SI = High word of size of region
Word at top of stack = Low word of size of region
BX = file handle
Outputs: Carry set + error code in AX if error encountered

Subfunction 0Ch
Get Disk Space
Inputs: ES:DI -> CDS for drive
Outputs: AL = Sectors per cluster
BX = Total clusters
CX = Bytes per sector
DX = Number of available clusters
Note: These are DOS preferred units. It is sufficient to return numbers such that (AL*BX*CX) accurately reflects the space available.

Subfunction 0Eh
Set File Attributes
Inputs: SDA.FN1 = Fully qualified filename
SDA.CURR_CDS = CDS for drive with file
Word at top of stack = New file attributes

Outputs: Carry set + error code in AX if error encountered

Subfunction 0Fh
Get File Attributes
Inputs: SDA.FN1 = Fully qualified filename
SDA.CURR_CDS = CDS for drive with file
Outputs: Carry set + error code in AX if error encountered
If no error, AX = file attributes

Subfunction 11h
Rename File
Inputs: SDA.FN1 = Current fully qualified filename
SDA.FN2 = New fully qualified filename
SDA.CURR_CDS = CDS for drive with file
Outputs: Carry set + error code in AX if error encountered

Subfunction 13h
Delete File
Inputs: SDA.FN1 = Fully qualified filespec (may contain wildcards)
SDA.CURR_CDS = CDS for drive with file
Outputs: Carry set + error code in AX if error encountered

Subfunction 16h
Open Existing File
Inputs: SDA.FN1 = Fully qualified filename
SDA.OPEN_MODE = Open mode for file
ES:DI -> Uninitialized SFT for the file
Outputs: Carry set + error code in AX if error encountered
SFT completed if no error
Note: Do not set the C_HANDLES field in the SFT. DOS maintains this field itself.

Subfunction 17h
Create/Truncate File
Inputs: SDA.FN1 = Fully qualified filename
ES:DI -> Uninitialized SFT for the file
SDA.CURR_CDS = CDS for drive with file
Word at top of stack = File attribute for file
Outputs: Carry set + error code in AX if error encountered
SFT completed if no error
Notes: It is known that this function is called by DOS function 5Bh. In order for 5Bh to fail if the file already exists, this function

must be able to communicate the pre-existence of a file via a register or SDA field. It is not known how this is done. Do not set the C_HANDLES field in the SFT. DOS maintains this field itself.

Subfunction 1Bh
Find First Matching File
Inputs: SDA.FN1 = Fully qualified filespec for search
SDA.SDB = Unitialized Search Data Block
SDA.CURR_DTA -> Directory info buffer for found file
SDA.SRCH_ATTR = Search attribute mask for file
Outputs: Carry set + error code in AX if error encountered
If no error, SDB initialized

Subfunction 1Ch
Find Next Matching File
Inputs: SDA.SDB = Search Data Block from last Find operation
SDA.CURR_DTA -> Directory info buffer for found file
Outputs: Carry set + AX = 12h if no more files

Subfunction 1Dh
Close all files for process
Inputs & Outputs: Mostly unknown. What is clear is that in order to implement this function, a record of all files opened by which processes on which machines is required to be maintained by the redirector.

Subfunction 1Eh
Do Redirection
Inputs: Word at top of stack = Command to execute
Other inputs depend on command to execute
Outputs: Carry set + error code in AX if error encountered
Other outputs depend on command to execute

Subfunction 1Fh
Printer Setup
Inputs: Word at top of stack = Command to execute
Other inputs depend on command to execute
Outputs: Carry set + error code in AX if error encountered
Other outputs depend on command to execute

Subfunction 20h
Flush All Disk Buffers
Inputs & Outputs: Mostly unknown

Subfunction 21h
Seek From End of File
Inputs: ES:DI -> SFT for file
CX:DX = Offset from end of file to position to
Outputs: Carry set + error code in AX if error encountered

Subfunction 22h
Process Termination Hook
Inputs & Outputs: Mostly unknown

Subfunction 23h
Qualify Path and Filename
Inputs: DS:SI -> Unqualified filename
ES:DI -> Buffer for fully qualified filename
Outputs: Carry set + error code in AX if error encountered
Note: DOS appears to supply a default name qualification function that does a very adequate job of this without a redirector needing to support it. The presence of a redirector appears only to be necessary should there be some form of directory or filename translation required. The output of this function, or the DOS default routine, is used directly to supply the input for the directory manipulation and file open/rename/delete etc. subfunctions.

Subfunction 25h
Redirected Printer Mode
Inputs: Word at top of stack = Command to execute
Other inputs depend on command to execute
Outputs: Carry set + error code in AX if error encountered
Other outputs depend on command to execute

Subfunction 2Eh
Extended Open File
Inputs: SDA.FN1 = Fully qualified filename
ES:DI -> Uninitialized SFT for the file

	Word at top of stack = File attribute for created/truncated file
	SDA.SPECOPEN_ACT = Action codes
	SDA.SPECOPEN_MODE = Open mode for file
Outputs:	Carry set + error code in AX if error encountered
	SFT completed if no error
Notes:	This was introduced with DOS 4.0 to provide a unified interface to the functionality supplied by Subfunction 16h 17h, and to support DOS Function 6Ch. That function returns to the caller a result field in CX indicating whether the file existed or not, and whether it has been truncated. In order for 6Ch to be able to return the appropriate result code, this function must be able to communicate the result via a register or SDA field. Currently, It is not known how this is done. Do not set the C_HANDLES field in the SFT. DOS maintains this field itself.

How Do We Know The Call's For Us?

Since there may be a chain of redirectors, each wanting to service only those calls that relate to the drive that it is redirecting, there must be a way of determining that a particular redirector call making the rounds is for you. There are two ways of doing this, depending on the type of operation to be performed. If the operation is one that deals with an open file, such as Read, Write, Commit, or Close, then ES:DI on entry points at the SFT entry for the file. The device information word in the SFT entry contains the drive number of the file device in the bottom six bits, so it is a simple masking and comparison operation to ascertain that the call is for you. If the operation does not deal with an open file, the SDA.DRIVE_CDSPTR field points at the CDS entry for whatever drive is being accessed during the redirector call. The most reliable way to compare the CDS entries is to match the characters in their CURR_PATH fields up to the root offset. Since this involves only a very few characters, it is also fairly efficient as well.

Example Program: The Phantom

In order to study the interface in detail, it is best to use a real example, albeit an example with limited usefulness. So here is the Phantom, the world's least effec-

tive storage device! The Phantom implements a phantom drive that supports DOS file system commands. Let us briefly set out its specification:

- It supports all the major DOS commands, in some capacity, including: DIR, MD, CD, RD, COPY, DEL, RENAME, ATTRIB, and VOL. In addition, it is possible to run a program from the phantom drive, if you can find one that's small enough (we used DEVLOD from chapter 3).
- It works under DOS versions in the range 3.10 to 5.00 inclusive (although in practice with some exceptions under DOS 4.0).
- In the interests of simplicity of implementation, limitations include: a single file can exist on the drive, although it can exist in any directory; the file can be a maximum of 2,048 bytes in size; each directory, including the root, can only have a single subdirectory.
- It is unloadable.

The Phantom is written in Turbo Pascal, but while a knowledge of the language will help understand the particular implementation techniques used, the source is heavily commented, and the principles are easily portable to C or assembly language.

How is this specification implemented against the redirector interface as you currently understand it? A small machine code interrupt handler stub intercepts all INT 2Fh calls. If not a redirector function call, the stub passes control on to the next handler in the INT 2Fh chain. If it is a redirector call, it passes control to the main redirector procedure that establishes, on the basis of the criteria outlined below, if the call refers to the drive that you are redirecting. If it does not, control is returned to the stub, which passes control on to the next handler in the chain. If the call is for "your" drive, the main redirector procedure calls the appropriate procedure to carry out the requested subfunction. When control finally returns to the stub, it returns back to DOS.

The main routine establishes whether the call is one you support, and whether it is for your drive. If it is, it prepares a register set and calls the appropriate routine to perform the requested subfunction. When the subfunction returns, having updated any register contents, the main routine reinstates the registers and returns. The code for the Phantom follows (PHANTOM.PAS; see the updated version on disk in CHAP4\PHANTOM.PAS):

```
{$A-,B-,D+,L+,E-,F-,I-,N-,O-,R-,S-,V-}
{$M 2048,128,1000}
program phantom_drive;
uses
```

```
    dos, crt;

type
    sig_rec = record
        signature : string[7];
        psp : word;
        drive_no : byte;
    end;

const
    cds_id_size = 10;
    cds_id = 'Phantom. :\';
    our : sig_rec =
        (   signature : 'PHANTOM'; psp : 0; drive_no : 0);
    vollab : string[13] = 'AN ILLUS.ION'#0; { Our Volume label }
    maxfilesize = 2047;                     { for our 1 file }

    isr_CODE_max = 102;                     { offset of last byte }
                                            { in our ISR macine code }

type
    strptr = ^string;
    cdsidarr = array[1..cds_id_size] of char;
    cdsidptr = ^cdsidarr;

{ FindFirst/Next data block - ALL DOS VERSIONS }
    sdb_ptr = ^sdb_rec;
    sdb_rec = record
        drv_lett : byte;
        srch_tmpl : array[0..10] of char;
        srch_attr : byte;
        dir_entry : word;
        par_clstr : word;
        f1 : array[1..4] of byte;
    end;

{ DOS System File Table entry - ALL DOS VERSIONS }
    sft_ptr = ^sft_rec;
    sft_rec = record
        handle_cnt,
        open_mode : word;
        attr_byte : byte;
        dev_info : word;
        devdrv_ptr : pointer;
        start_clstr,        { we don't need to touch this }
        f_time,
        f_date : word;
        f_size,
        f_pos : longint;
        rel_lastclstr,      { we don't need to touch this }
        abs_lastclstr,      { we don't need to touch this }
        dir_sector : word;  { we don't need to touch this }
        dir_entryno : byte; { we don't need to touch this }
```

```
        fcb_fn : array[0..10] of char;
    end;

{ DOS Current directory structure - DOS VERSION 3.xx }
    cds3_rec = record
        curr_path : array[0..66] of char;
        flags : word;
        f1 : array[1..10] of byte;  { we don't need to touch this }
        root_ofs : word;
    end;

{ DOS Current directory structure - DOS VERSION 4.xx }
    cds4_rec = record
        curr_path : array[0..66] of char;
        flags : word;
        f1 : array[1..10] of byte;  { we don't need to touch this }
        root_ofs : word;
        f2 : array[1..7] of byte;   { we don't need to touch this }
    end;

{ DOS Directory entry for 'found' file - ALL DOS VERSIONS }
    dir_ptr = ^dir_rec;
    dir_rec = record
        fname : array[0..10] of char;
        fattr : byte;
        f1 : array[1..10] of byte;
        time_lstupd,
        date_lstupd,
        start_clstr : word;         { we don't need to touch this }
        fsiz : longint;
    end;

{ Swappable DOS Area - DOS VERSION 3.xx }
    sda3_rec = record
         f0 : array[1..12] of byte;
         curr_dta : pointer;
         f1 : array[1..30] of byte;
         dd,
         mm : byte;
         yy_1980 : word;
         f2 : array[1..96] of byte;
         fn1,
         fn2 : array[0..127] of char;
         sdb : sdb_rec;
         found_file : dir_rec;
         drive_cdscopy : cds3_rec;
         fcb_fn1 : array[0..10] of char;
         f3 : byte;
         fcb_fn2 : array[0..10] of char;
         f4 : array[1..11] of byte;
         srch_attr : byte;
         open_mode : byte;
         f5 : array[1..48] of byte;
```

```
        drive_cdsptr : pointer;
        f6 : array[1..12] of byte;
        fn1_csofs,
        fn2_csofs : word;
        f7 : array[1..56] of byte;
        ren_srcfile : sdb_rec;
        ren_file : dir_rec;
    end;

{ Swappable DOS Area - DOS VERSION 4.xx }
    sda4_ptr = ^sda4_rec;
    sda4_rec = record
        f0 : array[1..12] of byte;
        curr_dta : pointer;
        f1 : array[1..32] of byte;
        dd,
        mm : byte;
        yy_1980 : word;
        f2 : array[1..106] of byte;
        fn1,
        fn2 : array[0..127] of char;
        sdb : sdb_rec;
        found_file : dir_rec;
        drive_cdscopy : cds4_rec;
        fcb_fn1 : array[0..10] of char;
        f3 : byte;
        fcb_fn2 : array[0..10] of char;
        f4 : array[1..11] of byte;
        srch_attr : byte;
        open_mode : byte;
        f5 : array[1..51] of byte;
        drive_cdsptr : pointer;
        f6 : array[1..12] of byte;
        fn1_csofs,
        fn2_csofs : word;
        f7 : array[1..71] of byte;
        spop_act,
        spop_attr,
        spop_mode : word;
        f8 : array[1..29] of byte;
        ren_srcfile : sdb_rec;
        ren_file : dir_rec;
    end;

{ DOS List of lists structure - DOS VERSIONS 3.1 thru 4 }
    lol_rec = record
        f1 : array[1..22] of byte;
        cds : pointer;
        f2 : array[1..7] of byte;
        last_drive : byte;
    end;

{ This serves as a list of the function types that we support }
```

```
    fxn_type = (_inquiry, _rd, _md, _cd, _close, _commit, _read,
                _write, _lock, _unlock, _space, _setattr, _getattr,
                _rename, _delete, _open, _create, _ffirst, _fnext,
                _seek, _specopen, _unsupported);

{ A de rigeur structure for manipulators of pointers }
    os = record o,s:word; end;

    fcbfnbuf = array[0..12] of char;
    fcbfnptr = ^fcbfnbuf;

    ascbuf = array[0..127] of char;
    ascptr = ^ascbuf;

{ This defines a pointer to our primary Int 2Fh ISR structure }
    isrptr = ^isr_rec;

{ A structure to contain all register values. The TP DOS registers
    type is insufficient }
    regset = record
        bp,es,ds,di,si,dx,cx,bx,ax,ss,sp,cs,ip,flags:word; end;

{ Our Int 2F ISR structure }
    isr_CODE_buffer = array[0..isr_CODE_max] of byte;
    isr_rec = record
        ic:isr_CODE_buffer;  { Contains our machine code ISR stub code }
        save_ss,             { Stores SS on entry before stack switch }
        save_sp,             { Stores SP on entry before stack switch }
        real_fl,             { Stores flags as they were on entry }
        save_fl,             { Stores flags from the stack }
        save_cs,             { Stores return CS from the stack }
        save_ip : word;      { Stores return IP from the stack }
        our_drive : boolean; { For ISR to either chain on or return }
    end;

    strfn = string[12];

const
 { all the calls we need to support are in the range 0..33 }
    fxn_map_max = $2e;
    fxn_map : array[0..fxn_map_max] of fxn_type =
                (_inquiry, _rd, _unsupported, _md, _unsupported,
                _cd, _close, _commit, _read, _write,
                _lock, _unlock, _space, _unsupported, _setattr,
                _getattr, _unsupported, _rename, _unsupported,
                _delete, _unsupported, _unsupported, _open, _create,
                _unsupported, _unsupported, _unsupported, _ffirst, _fnext,
                _unsupported, _unsupported, _unsupported, _unsupported,
                _seek, _unsupported, _unsupported, _unsupported,
                _unsupported, _unsupported, _unsupported, _unsupported,
                _unsupported, _unsupported, _unsupported, _unsupported,
                _unsupported, _specopen
                );
```

```
{ The following are offsets into the ISR stub code where run time
  values must be fixed in }
    prev_hndlr  = 99;
    redir_entry = 49;
    our_sp_ofs  = 45;
    our_ss_ofs  = 40;

{ The following offsets are known at compile time and are directly
  referenced in the ISR stub code }
    save_ss_ofs = isr_CODE_max+1;
    save_sp_ofs = isr_CODE_max+3;
    save_rf_ofs = isr_CODE_max+5;
    save_fl_ofs = isr_CODE_max+7;
    save_cs_ofs = isr_CODE_max+9;
    save_ip_ofs = isr_CODE_max+11;
    our_drv_ofs = isr_CODE_max+13;

{ Our ISR stub code is defined as a constant array of bytes which
  actually contains machine code as commented on the right }
    isr_CODE : isr_CODE_buffer = { entry: }
    (       $90,                { nop OR int 3          ; for debugging }
            $9c,                { pushf                 ; save flags    }
        $80,$fc,$11,            { cmp   ah,11h          ; our fxn?      }
        $75,$5a,                { jne   not_ours        ; bypass        }
    $2e,$8f,$06, save_rf_ofs, 0,{ pop   cs:real_fl      ; store act flgs}
    $2e,$8f,$06, save_ip_ofs, 0,{ pop   cs:save_ip      ; store cs:ip   }
    $2e,$8f,$06, save_cs_ofs, 0,{ pop   cs:save_cs      ; and flags     }
    $2e,$8f,$06, save_fl_ofs, 0,{ pop   cs:save_fl      ; from stack    }

    $2e,$89,$26, save_sp_ofs, 0,{ mov   cs:save_sp,sp   ; save stack    }
        $8c,$d4,                { mov   sp,ss                           }
    $2e,$89,$26, save_ss_ofs, 0,{ mov   cs:save_ss,sp                   }

        $bc,     0,0,           { mov   sp,SSEG         ; set our stack }
        $8e,$d4,                { mov   ss,sp                           }
        $bc,     0,0,           { mov   sp,SPTR                         }

        $9c,                    { pushf                 ; call our      }
        $9a,     0,0,0,0,       { call  redir           ; intr proc.    }

    $2e,$8b,$26, save_ss_ofs, 0,{ mov   sp,cs:save_ss   ; put back      }
        $8e,$d4,                { mov   ss,sp           ; caller's stack}
    $2e,$8b,$26, save_sp_ofs, 0,{ mov   sp,cs:save_sp                   }

    $2e,$ff,$36, save_fl_ofs, 0,{ push  cs:save_fl      ; restore       }
    $2e,$ff,$36, save_cs_ofs, 0,{ push  cs:save_cs      ; restore       }
    $2e,$ff,$36, save_ip_ofs, 0,{ push  cs:save_ip      ; return addr.  }
    $2e,$ff,$36, save_rf_ofs, 0,{ push  cs:real_fl      ; save act flgs }

    $2e,$80,$3e, our_drv_ofs,0,0,{ cmp cs:our_drive,0; not our drive?}
        $74,$04,                { je    not_ours        ; no, jump      }
        $9d,                    { popf                  ; yes, restore  }
        $ca,$02,$00,            { retf  2               ; & return flags}
```

```
                                { not_ours: }
        $9d,                        { popf                  ; restore flags }
        $ea,    0,0,0,0             { jmp   far prev_hndlr  ; pass the buck }
        );

var
{ The instance of our Int 2F ISR }
    isr : isrptr;

{ variables relating to the one allowable file.. }
    file_name : fcbfnbuf;
    file_buffer : array[0..maxfilesize] of byte;
    file_opens,
    file_date,
    file_time : word;
    file_attr : byte;
    file_size : longint;

{ Our full directory structure }
    max_path : ascbuf;

{ Global stuff }
    our_sp : word;          { SP to switch to on entry }
    dos_major,              { Major DOS vers }
    dos_minor,              { Minor DOS vers }
    drive_no : byte;        { A: is 1, B: is 2, etc. }
    strbuf : string;        { General purpose pascal string buffer }
    a1,                     { Pointer to an ASCIIZ string }
    a2 : ascptr;            { Pointer to an ASCIIZ string }
    drive : string[3];      { Command line parameter area }
    fxn : fxn_type;         { Record of function in progress }
    r : regset;             { Global save area for all caller's regs }
    temp_name : fcbfnbuf;   { General purpose ASCIIZ filename buffer }
    iroot,                  { Index to root directory in max_path }
    icur,                   { Index to current directory in max_path }
    lmax,                   { Length of max_path }
    ifile : byte;           { Index to directory in max_path with file }
    ver : word;             { full DOS version }
    sda : pointer;          { pointer to the Swappable Dos Area }
    lol : pointer;          { pointer to the DOS list of lists struct }

const h:array[0..15] of char = '0123456789abcdef';
type str4 = string[4];
function hex(inp:word):str4;
begin
    hex[0]:=#4;
    hex[1]:=h[inp shr 12];
    hex[2]:=h[(inp shr 8) and $f];
    hex[3]:=h[(inp shr 4) and $f];
    hex[4]:=h[inp and $f];
end;

{ Fail PHANTOM, print message, exit to DOS }
```

```
procedure failprog(msg:string);
begin
    writeln(msg);
    Halt(1);
end;

{ Get DOS version, address of Swappable DOS Area, and address of
  DOS List of lists. We only run on versions of DOS >= 3.10, so
  fail otherwise }
procedure get_dos_vars;
var r : registers;
begin
    ver:=dosversion;
    dos_major:=lo(ver);
    dos_minor:=hi(ver);
    if (dos_major<3) or ((dos_major=3) and (dos_minor<10)) then
        failprog('DOS Version must be 3.10 or greater');
    with r do
        begin
            ax:=$5d06; msdos(r); sda:=ptr(ds,si);   { Get SDA pointer }
            ax:=$5200; msdos(r); lol:=ptr(es,bx);   { Get LoL pointer }
        end;
end;

{ Fail the current redirector call with the supplied error number, i.e.,
  set the carry flag in the returned flags, and set ax=error code }
procedure fail(err:word);
begin
    r.flags:=r.flags or fcarry;
    r.ax:=err;
end;

{ Convert an 11 byte fcb style filename to ASCIIZ name.ext format }
procedure fnfmfcbnm(var ss; var p:ascptr);
var i,j:byte; s:ascbuf absolute ss;
    dot : boolean;
begin
    p:=@temp_name;
    i:=0;
    while (i<8) and (s[i]<>' ') do inc(i);
    move(s,p^,i);
    j:=8;
    while (j<11) and (s[j]<>' ') do inc(j);
    move(s,p^[succ(i)],j-8);
    if j<>8 then begin p^[i]:='.'; p^[j]:=#0; end
    else p^[i]:=#0;
end;

{ The opposite of the above, convert an ASCIIZ name.ext filename
  into an 11 byte fcb style filename }
procedure cnvt2fcb(var ss; var pp);
var i,j:byte;
    s:ascbuf absolute ss;
```

```
    p:ascbuf absolute pp;
begin
    i:=0; j:=0;
    fillchar(p,11,' ');
    while s[i]<>#0 do
        begin
            if s[i]='.' then j:=7 else p[j]:=s[i];
            inc(i);
            inc(j);
        end;
end;

{ Get the length of an ASCIIZ string }
function asclen(var a:ascbuf):word;
var i:word;
begin i:=0; while (i<65535) and (a[i]<>#0) do inc(i); asclen:=i; end;

{ Translate a maximum of strlim bytes of an ASCIIZ string to a Pascal string }
procedure ascii2string(src, dst : pointer; strlim : byte);
var i:integer;
begin
    byte(dst^):=strlim;
    move(src^,pointer(succ(longint(dst)))^,strlim);
    i:=pos(#0,string(dst^));
    if i<>0 then byte(dst^):=pred(i);
end;

{ Set up global a1 to point to the appropriate source for the file
  or directory name parameter for this call }
procedure set_fn1;
begin
    case fxn of
{ For these calls, a fully qualified file/directory name is given in the
  SDA first filename field. This field, incidentally, can also be referenced
  indirectly through the SDA first filename offset field into DOS's CS. }
        _rd .. _cd, _setattr .. _create, _ffirst, _specopen :
            if dos_major=3 then
                a1:=@sda3_rec(sda^).fn1
            else
                a1:=@sda4_rec(sda^).fn1;

{ These do not need a filename... }
        _close .. _write, _seek : ;

{ For findnext, an fcb style filename template is available within the
  SDA search data block field }
        _fnext :
            if dos_major=3 then
                a1:=@sda3_rec(sda^).sdb.srch_tmpl
            else
                a1:=@sda4_rec(sda^).sdb.srch_tmpl;
    end;
end;
```

```
{ Back up a directory level, i.e., go back to the previous \ in a path string }
function back_1(var path:ascbuf; var i:byte):boolean;
begin
    if i=iroot then begin back_1:=false; exit; end;
    repeat dec(i) until (i=iroot) or (path[i]='\');
    back_1:=true;
end;

{ Check that the qualified pathname that is in a1 matches our full
  directory structure to length lsrc. If not, fail with 'Path not found' }
function process_path(a1 : ascptr; lsrc : byte):boolean;
var isrc : byte;
begin
    process_path:=false;
    isrc:=0;
    for isrc:=0 to pred(lsrc) do
        if (isrc>lmax) or
            (a1^[isrc]<>max_path[isrc]) then
                begin fail(3); exit; end;
    inc(isrc);
    if max_path[isrc]<>'\' then fail(3)
    else process_path:=true;
end;

function the_time:word;
    function ticks:longint;
        { mov ah,0   int 1ah   mov ax,dx   mov dx,cx }
        inline($b4/$00/$cd/$1a/$8b/$c2/$8b/$d1);
var t:longint;
    hh, mm, s2 : word;
begin
    t:=ticks;
    hh:=t div (182*6*60);
    dec(t,hh*(182*6*60));
    mm:=t div (182*6);
    dec(t,mm*(182*6));
    s2:=(t*10) div 364;
    the_time:=(hh shl 11) or (mm shl 5) or s2;
end;

function the_date:word;
begin
    if dos_major=3 then
        with sda3_rec(sda^) do
            the_date:=(yy_1980 shl 9) or (mm shl 5) or dd
    else
        with sda4_rec(sda^) do
            the_date:=(yy_1980 shl 9) or (mm shl 5) or dd;
end;

{ Change Directory - subfunction 05h }
procedure cd;
var lsrc : byte;
```

```
begin
    lsrc:=asclen(a1^);
    if lsrc=succ(iroot) then dec(lsrc); { Special case for root }
    if not process_path(a1,lsrc) then exit;
    if dos_major=3 then             { Copy in the new path into the CDS }
        move(max_path,cds3_rec(sda3_rec(sda^).drive_cdsptr^).curr_path,lsrc)
    else
        move(max_path,cds4_rec(sda4_rec(sda^).drive_cdsptr^).curr_path,lsrc);
    icur:=lsrc;
end;

{ Remove Directory - subfunction 01h }
procedure rd;
var lsrc : byte;
begin
    lsrc:=asclen(a1^);
    if not process_path(a1,lsrc) then exit;
    if lsrc=icur then begin fail(5); exit; end;
    if lsrc=ifile then begin fail(5); exit; end;
    if lsrc<>lmax then begin fail(5); exit; end;
    if not back_1(max_path,lmax) then begin fail(3); exit; end;
    max_path[succ(lmax)]:=#0;
end;

{ Make Directory - subfunction 03h }
procedure md;
var lsrc, isrc : byte;
begin
    lsrc:=asclen(a1^);
    isrc:=lsrc;
    if not back_1(a1^,isrc) then begin fail(5); exit; end;
    if not process_path(a1,isrc) then exit;
    if isrc<>lmax then begin fail(5); exit; end;
    move(a1^,max_path,lsrc);
    max_path[lsrc]:='\';
    max_path[succ(lsrc)]:=#0;
    lmax:=lsrc;
end;

{ Close File - subfunction 06h }
procedure clsfil;
begin
{ Clear down supplied SFT entry for file }
    with sft_rec(ptr(r.es,r.di)^) do
        begin
            if file_opens=0 then begin fail(5); exit; end;
            dec(file_opens);
            if boolean(open_mode and 3) and
               not boolean(dev_info and $40) then
                begin                { if new or updated file... }
                    if f_date=0 then file_date:=the_date
                    else file_date:=f_date;
                    if f_time=0 then file_time:=the_time
```

```
                    else file_time:=f_time;
                end;
        end;
end;

{ Commit File - subfunction 07h }
procedure cmmtfil;
begin
{ We support this but don't do anything... }
    if file_opens=0 then fail(5);
end;

{ Read from File - subfunction 08h }
procedure readfil;
begin
    if file_opens=0 then begin fail(5); exit; end;

{ Fill the user's buffer (the DTA) from our internal; file buffer,
  and update the suplied SFT for the file }
    with sft_rec(ptr(r.es,r.di)^) do
        begin
            if (f_pos+r.cx)>f_size then r.cx:=f_size-f_pos;
            if dos_major=3 then
                move(file_buffer[f_pos],sda3_rec(sda^).curr_dta^,r.cx)
            else
                move(file_buffer[f_pos],sda4_rec(sda^).curr_dta^,r.cx);
            inc(f_pos,r.cx);
        end;
end;

{ Write to File - subfunction 09h }
procedure writfil;
begin
    if file_opens=0 then begin fail(5); exit; end;

{ Update our internal file buffer from the user buffer (the DTA) and
  update the supplied SFT entry for the file }
    with sft_rec(ptr(r.es,r.di)^) do
        begin
            if boolean(file_attr and readonly) then
                begin fail(5); exit; end;
            if (f_pos+r.cx)>maxfilesize then r.cx:=maxfilesize-f_pos;
            if dos_major=3 then
                move(sda3_rec(sda^).curr_dta^,file_buffer[f_pos],r.cx)
            else
                move(sda4_rec(sda^).curr_dta^,file_buffer[f_pos],r.cx);
            inc(f_pos,r.cx);
            if f_pos>file_size then file_size:=f_pos;
            f_size:=file_size;
            dev_info:=dev_info and (not $40);
        end;
end;
```

```
{ Get Disk Space - subfunction 0Ch }
procedure dskspc;
begin
{ Our 'disk' has 1 cluster containing 1 sector of 2048 bytes, and ... }
    r.ax:=1;
    r.bx:=1;
    r.cx:=succ(maxfilesize);
{ ... its either all available or none! }
    r.dx:=ord(ifile=0);
end;

{ Set File Attributes - subfunction 0Eh }
procedure setfatt;
var lsrc, isrc : byte;
begin
    lsrc:=asclen(a1^);
    isrc:=lsrc;
    if not back_1(a1^,isrc) then begin fail(2); exit; end;
    if not process_path(a1,isrc) then exit;
    if isrc<>ifile then begin fail(2); exit; end;
    inc(isrc);
    fillchar(temp_name,13,#0);
    move(a1^[isrc],temp_name,lsrc-isrc);
    if temp_name<>file_name then begin fail(2); exit; end;
    if file_opens>0 then fail(5)
    else  file_attr:=byte(ptr(r.ss,r.sp)^);
end;

{ Get File Attributes - subfunction 0Fh }
procedure getfatt;
var lsrc, isrc : byte;
begin
    lsrc:=asclen(a1^);
    isrc:=lsrc;
    if not back_1(a1^,isrc) then begin fail(2); exit; end;
    if not process_path(a1,isrc) then exit;
    if isrc<>ifile then begin fail(2); exit; end;
    inc(isrc);
    fillchar(temp_name,13,#0);
    move(a1^[isrc],temp_name,lsrc-isrc);
    if temp_name<>file_name then begin fail(2); exit; end;
    if file_opens>0 then begin fail(5); exit; end;
    r.ax:=file_attr;
end;

{ Rename File - subfunction 11h }
procedure renfil;
var lsrc, isrc, isav, i : byte;
    dot:boolean;
begin
    if dos_major=3 then
        a2:=ptr(r.ss,sda3_rec(sda^).fn2_csofs)
    else
```

```
        a2:=ptr(r.ss,sda4_rec(sda^).fn2_csofs);
    lsrc:=asclen(a1^);
    isrc:=lsrc;
    if not back_1(a1^,isrc) then begin fail(3); exit; end;
    if not process_path(a1,isrc) then exit;
    if isrc<>ifile then begin fail(2); exit; end;
    inc(isrc);
    fillchar(temp_name,13,#0);
    move(a1^[isrc],temp_name,lsrc-isrc);
{ Check that the current filename matches ours }
    if temp_name<>file_name then begin fail(2); exit; end;
    if boolean(file_attr and $7) then begin fail(5); exit; end;
    if file_opens>0 then begin fail(5); exit; end;
    lsrc:=asclen(a2^);
    isrc:=lsrc;
    if not back_1(a2^,isrc) then begin fail(3); exit; end;
    if not process_path(a2,isrc) then exit;
    ifile:=isrc;
    inc(isrc);
{ Put in the new file name }
    fillchar(file_name,13,#0);
    move(a2^[isrc],file_name,lsrc-isrc);
end;

{ This procedure does a wildcard match from the mask onto the target and,
  if a hit, updates the search data block and found file areas supplied }
function match(var m, t; var s : sdb_rec; var d : dir_rec;
                d_e, p_c : word; s_a : byte) : boolean;
var i, j : byte;
    mask : ascbuf absolute m;
    tgt : ascbuf absolute t;
begin
    i:=0; j:=0;
    if tgt[0] in ['\',#0] then begin match:=false; exit; end;
    while i<11 do
        case mask[i] of
            '?' :   if tgt[j] in [#0,'\','.'] then
                        if (i=8) and (tgt[j]='.') then inc(j) else inc(i)
                    else
                        begin inc(i); inc(j); end;
            ' ' :   if tgt[j] in ['.','\',#0] then inc(i)
                    else begin match:=false; exit; end;
            else    if (i=8) and (tgt[j]='.') then inc(j)
                    else
                    if tgt[j]=mask[i] then begin inc(i); inc(j); end
                    else begin match:=false; exit; end;
        end;
    if not (tgt[j] in ['\',#0]) then begin match:=false; exit; end;
    with s do
        begin
            move(mask,srch_tmpl,11);
            dir_entry:=d_e;
            srch_attr:=s_a;
```

```
            par_clstr:=p_c;
            drv_lett:=drive_no or $80;
        end;
    with d do
        begin
            i:=0; j:=0;
            fillchar(fname,11,' ');
            while not (tgt[i] in [#0,'\']) do
                if tgt[i] = '.' then begin j:=8; inc(i); end
                else begin fname[j]:=tgt[i]; inc(i); inc(j); end;
            case d_e of
                0 : fattr:=$08;
                1 : fattr:=$10;
                2 : fattr:=file_attr;
            end;
            time_lstupd:=file_time;
            date_lstupd:=file_date;
            case d_e of
                0, 1 : fsiz:=0;
                2 : fsiz:=file_size;
            end;
        end;
    match:=true;
end;

{ Delete File - subfunction 13h }
procedure delfil;
var isrc, lsrc : byte;
    sdb:sdb_rec;    { These are dummies for the match procedure to hit }
    der:dir_rec;
begin
    lsrc:=asclen(a1^);
    isrc:=lsrc;
    if not back_1(a1^,isrc) then begin fail(3); exit; end;
    if not process_path(a1,isrc) then exit;
    if isrc<>ifile then begin fail(2); exit; end;

    inc(os(a1).o,succ(isrc));
    cnvt2fcb(a1^,temp_name);
    if ((file_attr and $1f)>0) then begin fail(5); exit; end;
    if not match(temp_name,file_name,sdb,der,0,0,0) then
        begin fail(2); exit; end;
    if file_opens=0 then ifile:=0 else fail(5);
end;

{ Open Existing File - subfunction 16h }
procedure opnfil;
var isrc, lsrc : byte;
begin
    lsrc:=asclen(a1^);
    isrc:=lsrc;
    if not back_1(a1^,isrc) then begin fail(3); exit; end;
    if not process_path(a1,isrc) then exit;
```

```
    if isrc<>ifile then begin fail(2); exit; end;
    inc(isrc);
    fillchar(temp_name,13,#0);
    move(a1^[isrc],temp_name,lsrc-isrc);
{ Check file names match }
    if temp_name<>file_name then begin fail(2); exit; end;

{ Initialize supplied SFT entry }
    with sft_rec(ptr(r.es,r.di)^) do
        begin
            file_attr:=byte(ptr(r.ss,r.sp)^);
            if dos_major=3 then
                open_mode:=sda3_rec(sda^).open_mode and $7f
            else
                open_mode:=sda4_rec(sda^).open_mode and $7f;
            cnvt2fcb(temp_name,fcb_fn);
            inc(file_opens);
            f_size:=file_size;
            f_date:=file_date;
            f_time:=file_time;
            dev_info:=$8040 or drive_no; { Network drive, unwritten to }
            dir_sector:=0;
            dir_entryno:=0;
            attr_byte:=file_attr;
            f_pos:=0;
            devdrv_ptr:=nil;
        end;
end;

{ Truncate/Create File - subfunction 17h }
procedure creatfil;
var isrc, lsrc : byte;
begin
    lsrc:=asclen(a1^);
    isrc:=lsrc;
    if not back_1(a1^,isrc) then begin fail(3); exit; end;
    if not process_path(a1,isrc) then exit;

    if ifile=0 then
        begin
{ Creating new file }
            ifile:=isrc;
            inc(isrc);
            if isrc=lsrc then begin fail(13); ifile:=0; exit; end;
            fillchar(file_name,13,#0);
            move(a1^[isrc],file_name,lsrc-isrc);
        end
    else

    if ifile=isrc then
        begin
{ Truncate existing file }
            inc(isrc);
```

```
            fillchar(temp_name,13,#0);
            move(a1^[isrc],temp_name,lsrc-isrc);
            if temp_name<>file_name then begin fail(2); exit; end;
            if boolean(file_attr and $7) then begin fail(5); exit; end;
            if file_opens>0 then begin fail(5); exit; end;
        end
    else fail(82);  { This provokes a 'ran out of dir entries' error }

{ Initialize supplied SFT entry }
    with sft_rec(ptr(r.es,r.di)^) do
        begin
            file_attr:=byte(ptr(r.ss,r.sp)^); { File attr is top of stack }
            open_mode:=$01;     { assume an open mode, none is supplied.. }
            cnvt2fcb(file_name,fcb_fn);
            inc(file_opens);
            f_size:=0;
            f_pos:=0;
            file_size:=0;
            dev_info:=$8040 or drive_no; { Network drive, unwritten to }
            dir_sector:=0;
            dir_entryno:=0;
            f_date:=0;
            f_time:=0;
            devdrv_ptr:=nil;
            attr_byte:=file_attr;
        end;
end;

{ Special Multi-Purpose Open File - subfunction 2Eh }
procedure spopnfil;
var isrc, lsrc : byte;
    action, mode, result : word;
begin
    lsrc:=asclen(a1^);
    isrc:=lsrc;
    if not back_1(a1^,isrc) then begin fail(3); exit; end;
    if not process_path(a1,isrc) then exit;
    mode:=sda4_rec(sda^).spop_mode and $7f;
    action:=sda4_rec(sda^).spop_act;
{ First, check if file must or must not exist }
    if ((((action and $f)=0) and (isrc<>0)) or
        (((action and $f0)=0) and (isrc=0))) then begin fail(5); exit; end;

    if ifile=0 then
        begin
{ Creating new file }
            result:=2;
            ifile:=isrc;
            inc(isrc);
            if isrc=lsrc then begin fail(13); ifile:=0; exit; end;
            fillchar(file_name,13,#0);
            move(a1^[isrc],file_name,lsrc-isrc);
        end
```

```
    else

    if ifile=isrc then
        begin
{ Open/Truncate existing file }
            inc(isrc);
            fillchar(temp_name,13,#0);
            move(a1^[isrc],temp_name,lsrc-isrc);
            if temp_name<>file_name then begin fail(82); exit; end;
            if boolean(action and 2) then
                result:=3           { File existed, was replaced }
            else
                result:=1;          { File existed, was opened }
            if boolean(file_attr and $1) and
                ((result=3) or ((mode and 3)>0)) then
                begin fail(5); exit; end;   { It's a read only file }
            if (result=3) and (file_opens>0) then
                begin fail(5); exit; end;   { Truncating an open file }
        end
    else fail(5);

{ Initialize the supplied SFT entry }
    with sft_rec(ptr(r.es,r.di)^) do
        begin
            if result>1 then
                begin
                    file_attr:=byte(ptr(r.ss,r.sp)^); { Attr is top of stack }
                    f_size:=0;
                    file_size:=0;
                end;
            open_mode:=mode;
            cnvt2fcb(file_name,fcb_fn);
            inc(file_opens);
            f_pos:=0;
            f_date:=0;
            f_time:=0;
            dev_info:=$8040 or drive_no; { Network drive, unwritten to }
            dir_sector:=0;
            dir_entryno:=0;
            devdrv_ptr:=nil;
            attr_byte:=file_attr;
        end;
end;

{ FindFirst - subfunction 1Bh }
procedure ffirst;
var isrc, lsrc : byte;
    sdb : sdb_ptr;
    der : dir_ptr;
    sa, fa : byte;
begin
    lsrc:=asclen(a1^);
    isrc:=lsrc;
```

```
    if not back_1(a1^,isrc) then begin fail(3); exit; end;
    if not process_path(a1,isrc) then exit;
    a2:=@max_path;
    if dos_major=3 then
        begin
            a1:=@sda3_rec(sda^).fcb_fn1;
            sdb:=@sda3_rec(sda^).sdb;
            der:=@sda3_rec(sda^).found_file;
            sa:=sda3_rec(sda^).srch_attr;
        end
    else
        begin
            a1:=@sda4_rec(sda^).fcb_fn1;
            sdb:=@sda4_rec(sda^).sdb;
            der:=@sda4_rec(sda^).found_file;
            sa:=sda4_rec(sda^).srch_attr;
        end;
    fa:=file_attr and $1e;
    inc(os(a2).o,succ(isrc));

{ First try and match volume label, if asked for }
    if ((sa=$08) or (boolean(sa and $08) and (isrc=iroot))) and
       match(a1^,vollab[1],sdb^,der^,0,isrc,sa) then exit;

{ Then try the one possible subdirectory, if asked for and if it exists }
    if boolean(sa and $10) and
       match(a1^,a2^,sdb^,der^,1,isrc,sa) then exit;

{ Finally try the one possible file, if asked for, if it exists, and if
  in this subdirectory }
    if (ifile=isrc) and
       ((fa=0) or boolean(sa and fa)) and
       match(a1^,file_name,sdb^,der^,2,isrc,sa) then exit;

{ Otherwise report no more files }
    fail(18);
end;

{ FindFirst - subfunction 1Bh }
procedure fnext;
var fa : byte;
    sdb : sdb_ptr; der : dir_ptr;
begin
    if dos_major=3 then
        begin
            sdb:=@sda3_rec(sda^).sdb;
            der:=@sda3_rec(sda^).found_file;
        end
    else
        begin
            sdb:=@sda4_rec(sda^).sdb;
            der:=@sda4_rec(sda^).found_file;
        end;
```

```
    fa:=file_attr and $1e;
    inc(sdb^.dir_entry);
    case sdb^.dir_entry of
        1 : a2:=@max_path[succ(sdb^.par_clstr)];
        2 : a2:=@file_name;
        else begin fail(18); exit; end;
    end;

{ First try the one possible subdirectory, if it exists. FNext can never
  match a volume label }
    if (sdb^.dir_entry=1) and boolean(sdb^.srch_attr and $10) and
        match(a1^,a2^,sdb^,der^,
            sdb^.dir_entry,sdb^.par_clstr,sdb^.srch_attr) then exit;

{ Then try the one possible file, if exists, and if in this subdirectory }
    if sdb^.dir_entry=1 then
        begin a2:=@file_name; sdb^.dir_entry:=2; end;
    if (sdb^.dir_entry=2) and (ifile=sdb^.par_clstr) and
        ((fa=0) or boolean(sdb^.srch_attr and fa)) and
        match(a1^,a2^,sdb^,der^,
            sdb^.dir_entry,sdb^.par_clstr,sdb^.srch_attr) then exit;

{ Otherwise return no more files }
    fail(18);
end;

{ Seek From End Of File - subfunction 21h }
procedure skfmend;
var skamnt : longint;
begin
    skamnt:=(longint(r.cx)*65536)+r.dx;
    if file_opens=0 then begin fail(5); exit; end;

{ Update supplied SFT entry for file }
    with sft_rec(ptr(r.es,r.di)^) do
        begin
            f_pos:=f_size-skamnt;
            r.dx:=f_pos shr 16;
            r.ax:=f_pos and $ffff;
        end;
end;

function call_for_us(es,di:word):boolean;
var p:pointer;
begin
    if (fxn in [_close.._unlock,_seek]) then
        call_for_us:=(sft_rec(ptr(es,di)^).dev_info and $1f)=drive_no
    else
    if fxn=_inquiry then call_for_us:=true
    else
        begin
            if dos_major=3 then p:=sda3_rec(sda^).drive_cdsptr
            else p:=sda4_rec(sda^).drive_cdsptr;
```

```
            call_for_us:=cdsidptr(p)^=cdsidptr(@max_path)^;
        end;
end;

{ This is the main entry point for the redirector. The procedure is actually
  invoked from the Int 2F ISR stub via a PUSHF and a CALL FAR IMMEDIATE
  instruction to simulate an interrupt.  That way we have many of the
  registers on the stack and DS set up for us by the TP interrupt keyword.
  This procedure saves the registers into the regset variable, assesses if
  the call is for our drive, and if so, calls the appropriate routine. On
  exit, it restores the (possibly modified) register values. }
procedure redirector(_flags,_cs,_ip,_ax,_bx,_cx,_dx,_si,_di,_ds,_es,_bp:word);
    interrupt;
begin
    with r do
        begin
            isr^.our_drive:=false;
{ If we don't support the call, pretend we didn't see it...! }
            if lo(_ax)>fxn_map_max then exit
            else fxn:=fxn_map[lo(_ax)];
            if fxn=_unsupported then exit;
{ If the call isn't for our drive, jump out here... }
            if not call_for_us(_es,_di) then exit;
{ Set up our full copy of the registers }
            isr^.our_drive:=true;
            move(_bp,bp,18); ss:=isr^.save_ss; sp:=isr^.save_sp;
            cs:=isr^.save_cs; ip:=isr^.save_ip; flags:=isr^.real_fl;
            ax:=0; flags:=flags and not fcarry;
            set_fn1;
            case fxn of
                _inquiry    : r.ax:=$00ff;
                _rd         : rd;
                _md         : md;
                _cd         : cd;
                _close      : clsfil;
                _commit     : cmmtfil;
                _read       : readfil;
                _write      : writfil;
                _space      : dskspc;
                _setattr    : setfatt;
                _lock, _unlock : ;
                _getattr    : getfatt;
                _rename     : renfil;
                _delete     : delfil;
                _open       : opnfil;
                _create     : creatfil;
                _specopen   : spopnfil;
                _ffirst     : ffirst;
                _fnext      : fnext;
                _seek       : skfmend;
            end;
{ Restore the registers, including any that we have modified.. }
            move(bp,_bp,18); isr^.save_ss:=ss; isr^.save_sp:=sp;
```

```
            isr^.save_cs:=cs; isr^.save_ip:=ip; isr^.real_fl:=flags;
        end;
end;

{ This procedure sets up our ISR stub as a structure on the heap. It
  also ensures that the structure is addressed from an offset of 0 so
  that the CS overridden offsets in the ISR code line up. Finally, it
  fixes in some values which are only available to us at run time,
  either because they are variable, or because of limitations of the
  language. }
procedure init_isr_CODE;
var p:pointer;
    i:pointer absolute isr;
begin
    getmem(isr,sizeof(isr_rec)+15);
    inc(os(isr).s,(os(isr).o+15) shr 4);
    isr^.ic:=isr_CODE;
    getintvec($2f,p);
    os(isr).o:=redir_entry; pointer(i^):=@redirector;
    os(isr).o:=our_ss_ofs; word(i^):=sseg;
    os(isr).o:=our_sp_ofs; word(i^):=our_sp;
    os(isr).o:=prev_hndlr; pointer(i^):=p;
    os(isr).o:=0;
end;

{ Do our initializations }
procedure init_vars;
    function installed_2f:byte;
        { mov ax,1100h   int 2fh }
        inline($b8/$00/$11/$cd/$2f);
begin
    if installed_2f=1 then
        failprog('Not OK to install a redirector...');
    drive_no:=byte(drive[1])-byte('@');
    our_sp:=sptr+$100;
    file_opens:=0;
{ Note that the assumption is that we lost 100h bytes of stack
  on entry to main }
{ Initialize and fix-up the master copy of the ISR code }
    init_isr_CODE;
    ifile:=0;
end;

{ This is where we do the initializations of the DOS structures
  that we need in order to fit the mould }
procedure set_path_entry;
var our_cds:pointer;
begin
    our_cds:=lol_rec(lol^).cds;
    if dos_major=3 then
        inc(os(our_cds).o,sizeof(cds3_rec)*pred(drive_no))
    else
        inc(os(our_cds).o,sizeof(cds4_rec)*pred(drive_no));
```

```
    if drive_no>lol_rec(lol^).last_drive then
        failprog('Drive letter higher than last drive...');

{ Edit the Current Directory Structure for our drive }
    with cds3_rec(our_cds^) do
        begin
            ascii2string(@curr_path,@strbuf,255);
            writeln('Curr path is ',strbuf);
            if (flags and $c000)<>0 then
                failprog('Drive already assigned.');
            flags:=flags or $c000;  { Network+Physical bits on ... }
            strbuf:=cds_id;
            strbuf[length(strbuf)-2]:=char(byte('@')+drive_no);
            move(strbuf[1],curr_path,byte(strbuf[0]));
            move(curr_path,max_path,byte(strbuf[0]));
            curr_path[byte(strbuf[0])]:=#0;
            max_path[byte(strbuf[0])]:=#0;
            root_ofs:=pred(length(strbuf));
            iroot:=root_ofs;
            lmax:=iroot;
        end;
end;

{ Use in place of Turbo's 'keep' procedure. It frees the environment
  and keeps the size of the TSR in memory smaller than 'keep' does }
procedure tsr;
var r:registers;
begin
    swapvectors;
    r.ax:=$4900;
    r.es:=memw[prefixseg:$2c];
    msdos(r);
    r.ax:=$3100;
    r.dx:=os(heapptr).s-prefixseg+1;
    msdos(r);
end;

procedure settle_down;
var p:pointer;
    i:integer;
    w:word;
begin
{ Plug ourselves into Int 2F }
    setintvec($2f,isr);
    writeln('Phantom drive installed as ',drive[1],':');
{ Find ourselves a free interrupt to call our own. Without it, future
  invocations of Phantom will not be able to unload us. }
    i:=$60;
    while (i<=$67) and (pointer(ptr(0,i shl 2)^)<>nil) do inc(i);
    if i=$68 then
        begin
            writeln('No user intrs available. PHANTOM not unloadable..');
            tsr;
```

```
        end;
{ Have our new found interrupt point at the command line area of
  our PSP. Complete our signature record, put it into the command line,
  and go to sleep. }
    w:=$80;
    setintvec(i,ptr(prefixseg,w));
    our.psp:=prefixseg;
    our.drive_no:=drive_no;
    sig_rec(ptr(prefixseg,w)^):=our;
    tsr;
end;

{ Find the latest Phantom installed, unplug it from the Int 2F chain if
  possible, undo the dpb chain, make the CDS reflect an invalid drive,
  and free its memory.. }
procedure do_unload;
var i:integer; p, cds:pointer; w:word; r:registers;
begin
    i:=$67;
    while (i>=$60) and
      (sig_rec(pointer(ptr(0,i shl 2)^)^).signature<>our.signature) do
        dec(i);
    if i=$5f then
        begin writeln(our.signature,' not found...'); halt; end;
    getintvec($2f,p);
    if os(p).o<>0 then
        failprog('2F superceded...');
    os(p).o:=prev_hndlr;
    setintvec($2f,pointer(p^));
    getintvec(i,p);
    drive_no:=sig_rec(p^).drive_no;
    with r do
        begin
            ax:=$4900; es:=sig_rec(p^).psp;
            msdos(r);
            if boolean(flags and fcarry) then
                writeln('Could not free main memory...');
        end;
    setintvec(i,nil);
    cds:=lol_rec(lol^).cds;
    if dos_major=3 then
        inc(os(cds).o,sizeof(cds3_rec)*pred(drive_no))
    else
        inc(os(cds).o,sizeof(cds4_rec)*pred(drive_no));
    with cds3_rec(cds^) do flags:=flags and $3fff;
    writeln('Drive ',char(byte('@')+drive_no),': is now invalid.');
end;

begin { MAIN }
{ Check parameter count }
    if (paramcount<>1) then
        failprog('Usage is: PHANTOM drive-letter:');
    drive:=paramstr(1);
```

```
    drive[1]:=upcase(drive[1]);
{ If this is an unload request, go to it }
    if (drive='-u') or (drive='-U') then
        begin
            get_dos_vars;
            do_unload;
            halt;
        end;
{ Otherwise, check that it's a valid drive letter }
    if  (length(drive)>2) or
        not (drive[1] in ['A'..'Z']) or
        ((length(drive)=2) and (drive[2]<>':'))
            then failprog('Usage is: PHANTOM drive-letter:');
{ ... and set up shop }
    init_vars;
    get_dos_vars;
    set_path_entry;
    settle_down;
end.
```

Here is a brief sample session with the Phantom drive:

```
C:\UNDOC>  phantom d:
Curr path is D:\
Phantom installed as D:
C:\UNDOC>  d:
D:\>  md test
D:\>  cd test
D:\TEST>  c:truename . > tmp.tmp
D:\TEST>  type tmp.tmp
Phantom.D:\TEST
D:\TEST>  dir

 Volume in drive D is AN ILLUSION
 Directory of  D:\TEST

TMP       TMP       17   9-15-90  19:52p
        1 File(s)          0 bytes free
```

Not very impressive looking at first glance, but all the elements of a full-blown file system are here. We have created an entity which looks like a drive, behaves like a drive, and yet which has no reality outside our INT 2Fh Function 11h handler. By the way, even undocumented DOS calls work properly on the redirected drive. This is another real advantage of using the redirector instead of hooking INT 21h. If you hook INT 21h, the only way that undocumented calls like Function 60h will work properly with your drive is if you write handlers for

these undocumented calls. But with the redirector interface, DOS takes care of cooking all file-system requests down for us.

Conclusion

In addition to talking a lot about the CDS, SFT, JFT, DPB, system FCBs, and so on, there seems to be one central point that emerges from this rather lengthy chapter: the DOS file system isn't just for plain old disks anymore. Any file system is primarily a logical rather than a physical construct, and DOS is no exception. In particular, commands like JOIN and SUBST, and the introduction of networking, have steadily moved the DOS file system away from the mundane world of cylinders/tracks/sectors, toward a more abstract notion of file store. A disk, directory, or file is simply anything that *acts* like one. The PC programmer who grasps DOS's support for non-FAT file systems can create many otherwise unimaginable programs.

Chapter 5

Memory Resident Software: Pop-ups and Multitasking

Raymond J. Michels, Tim Paterson, and Andrew Schulman

If there is any area of undocumented DOS with which PC programmers are generally familiar, it is writing memory-resident programs. Because such programs call the DOS Terminate and Stay Resident (TSR) function (INT 21h Function 31h) or the older TSR interrupt (INT 27h), they are often called TSRs. However, these documented facilities are insufficient for writing TSRs that, once resident, make INT 21h DOS calls. As noted in chapter 1, it is well known within the PC programming community that one must use undocumented DOS in order to properly write the vast majority of TSRs.

Given the continuing importance of memory-resident software in the PC marketplace, it is not surprising that much has been written about using undocumented DOS to write TSRs. Microsoft itself published a definitive piece on the subject, Richard Wilton's "Terminate-and-Stay-Resident Utilities," in the massive *MS-DOS Encyclopedia*. Wilton's article discusses the following undocumented DOS functions and interrupts:

- INT 21h Function 34h (Return InDOS Pointer)
- INT 21h Function 50h (Set PSP Segment)

- INT 21h Function 51h (Get PSP Segment)
- INT 21h Function 5D0Ah (Set Extended Error Information)
- INT 28h (Keyboard Busy Loop)

In addition, several books on C programming for the PC (see the bibliography in Appendix B of this book) include generic TSRs that use this same core set of undocumented DOS functions. The popular utilities published in each issue of *PC Magazine* are often TSRs whose assembly listings and prose descriptions show the intricacies of using these undocumented DOS functions.

Numerous commercial packages that provide generic TSR libraries are also available. These libraries use undocumented DOS, and so, by extension, do any applications built using them. Some of the commercial TSR libraries available are:

- CodeRunneR (for C or assembler; Microsystems Software, Framingham, MA)
- /*resident_C*/ (South Mountain Software, South Orange, NJ)
- C Tools Plus 6.0 (Blaise Computing, Berkeley, CA)
- Zortech C++ (includes TSR library; Zortech, Woburn, MA)
- Object Professional 1.0 (for Turbo Pascal 5.5; Turbo Power Software, Scotts Valley, CA)
- Magic TSR Toolkit (for ASM; Quantasm Corporation, Cupertino, CA)
- Dr. Switch-ASE ("Application Swapping Extensions" for dBase, FoxBase, Clipper; Black & White International, New York, NY)
- Stay-Res Plus (for BASIC; Micro Help)
- BATCOM (batch file compiler with TSR option; Wenham Software, Wenham, MA)

Is there anything new to say on this subject? Surprisingly, yes. Several areas of TSR programming with undocumented DOS have not been adequately covered elsewhere. These include:

- INT 21h Functions 5D06h, 5D0Bh (Get DOS Swappable Data Area)
- TSR termination
- Using Microsoft C (rather than Turbo C) interrupt functions
- Writing non-pop-up TSRs

This chapter presents a generic TSR skeleton for Microsoft C (versions 5.1 and 6.0), which you can use to "TSRify" your own programs. This generic TSR will be used to turn utilities from other parts of this book into "pop-ups" that are

activated by the press of a user-defined "hotkey" (we will discuss these terms in more detail in a moment).

The last section of this chapter presents a memory-resident program which is *not* activated by a hotkey; instead, it is periodically activated by the PC's timer tick, thereby multitasking in the background with whatever programs you run from the DOS command line in the foreground. The program is particularly interesting because it is an add-on to the PRINT multitasking TSR that comes with DOS.

TSR: It Sounds Like a Bug, But It's a Feature

Only three functions are absolutely necessary to write memory-resident software for MS-DOS; these three functions are fully documented:

- Terminate and Stay Resident (INT 21h Function 31h)
- Set Interrupt Vector (INT 21h Function 25h)
- Get Interrupt Vector (INT 21h Function 35h)

A TSR is any DOS program that calls INT 21h Function 31h (or its equivalent interrupt, INT 27h). The description of this function's purpose in the IBM DOS 3.3 *Technical Reference* is: "Terminates the current process and attempts to set the initial allocation block to the memory size in paragraphs." Doesn't sound too exciting. The TSR function is very much like the "normal" DOS termination function (INT 21h Function 4Ch), which kills off whatever program calls it, *except* that all memory belonging to the program is not released. Instead, part or all of the program's initial allocation block is reserved so that it will not be overlaid by the next program to be loaded.

Thus, a TSR is any DOS program that leaves bits of itself behind after terminating. This sounds like a classic bug (sometimes referred to as the "leaky bucket"), wherein memory is allocated but never gets deallocated. It doesn't sound like a feature around which an entire software industry could be built.

What is the advantage to terminating without freeing all your memory? If you've terminated, and some other program is now running, there's not much your memory is going to do other than take up space, right? True, chewing up memory can occasionally serve a purpose. In fact, TSRs have been written with names like MEMHOG and EATMEM to allow a developer with, say, a 640KB machine to test software under conditions similar to those on, say, a 512KB machine. But aside from this limited use, what good is it to hog memory after you're gone? You can't take it with you!

This is where the second necessary function, Set Interrupt Vector, comes in. All machines based on the Intel 80x86 architecture allow any program to install code that will get invoked whenever a hardware or software interrupt is generated. For example, the only reason INT 21h is a gateway to MS-DOS services is that interrupt vector 21h points to code inside DOS that provides these services. The ability to hang a piece of code off of an interrupt vector is what makes the TSR function something other than an elaborate way to consume memory. We can use the Set Interrupt Vector function to point interrupt vectors at our code, and then call the TSR function to keep this code (and associated data and stack space) resident in memory after we've terminated. Whenever one of our interrupts is generated, the code we've left behind will be activated. Thus, there really is life after termination; you *can* take it with you.

What sort of interrupts would a TSR be interested in trapping? The most obvious one is the hardware interrupt, INT 9, generated every time a user presses a key. By trapping INT 9, a TSR can watch every key that a user types. Let's say our TSR is a memory-resident Gilbert and Sullivan sampler that plays a selection from *The Mikado* whenever the user presses Alt-M, or *The Pirates of Penzance* whenever the user presses Alt-P. These are the only two keys we are interested in, and they are referred to as the program's hotkeys. Each time the user hits a key, the INT 9 handler wakes up, looks at the key, and, if it is not one of our hotkeys, goes back to sleep. But if it is one of our hotkeys, then our application should do its thing. In this example, this means playing light opera (Tarantara!), but in TSRs in general this sudden seeming springing to life is called the pop-up.

Now, one item has been glossed over: when the user types a key that is not one of our TSR's hotkeys, how does the key go to its true destination? Our TSR can't just discard it, but must somehow let other programs get a crack at it. It does this by calling whichever function *previously* owned the INT 9 vector, before our TSR installed its INT 9 handler. That means, *before* setting an interrupt vector, almost all TSRs will have to get its previous value, by calling the DOS Get Interrupt Vector function. Thus, our TSR looks something like this:

```
INTERRUPT PTR old_int9_handler;

INTERRUPT my_int9_handler()
    IF (key == alt_m)
        mikado();
    ELSE IF (key == alt_p)
        penzance();
```

```
    ELSE
        CALL PTR old_int9_handler();

BEGIN
    old_int9_handler = _dos_getvect(9); // INT 21h Function 35h
    _dos_setvect(9, my_int9_handler);   // INT 21h Function 25h
    go_tsr();                           // INT 21h Function 31h
```

If every program that has hooked INT 9 takes care to call the interrupt's previous owner, then every program that needs to will get a peek at the stream of user keystrokes. Calling the previous owner is known as chaining the interrupt and the end result is an interrupt chain: every time you press a key, a whole host of programs might see it. This mechanism for multiple-program access to the keyboard input stream was formalized in the OS/2 concept of the "monitor."

While the best-known TSRs (such as Borland's SideKick) are pop-ups that are activated by hotkeys, pressing a hotkey is just one way of generating an interrupt. Anything that generates an interrupt can be used to reactivate a TSR. For example, when our own program calls INT 21h, it's generating a software interrupt, so a TSR could easily attach itself to INT 21h, providing a mechanism for extending the operating system (or for debugging, as in the INTRSPY TSR in chapter 8). For example:

```
INTERRUPT PTR old_int21_handler;

INTERRUPT my_int21_handler()
    IF (ah == some function we're interested in)
        // do pre-processing
        // maybe CALL PTR old_int21_handler()
        // do post-processing
    ELSE
        CALL PTR old_int_21_handler();

BEGIN
    old_int21_handler = _dos_getvect(0x21);
    _dos_setvect(0x21, my_int21_handler);
    go_tsr();
```

In this example, as soon as we've attached my_handler to INT 21h by calling _dos_setvect, *all* INT 21h calls pass through the code in my_handler. This means that our own call to INT 21h Function 31h in go_tsr is actually first processed in my_handler. It is entirely up to the code in my_handler to determine what hap-

pens with each INT 21h request. Presumably the call to Function 31h would pass through unchanged to old_int21_handler, which might be MS-DOS or might be some other TSR that has hooked INT 21h, such as a command-line editor or network shell.

With all this power, it is essential that programs reserve the TSR facility for genuinely useful code that is worth having resident in memory. Software that helps the user prepare his last will and testament, for example, is not a good candidate for memory residency. Neither, for that matter, is our Gilbert and Sullivan sampler, since readily-available dedicated hardware already exists for this purpose.

Where Does Undocumented DOS Come In?

Since the three functions we need to produce the TSR are all fully documented, where does undocumented DOS come in? Do we really need undocumented DOS in order to write a program that plays "I Am the Very Model of a Modern Major General" whenever the user presses the Alt-P hotkey?

Unfortunately, we almost definitely do. Unless if we have achieved remarkable data compression, we will not want the notes for our music occupying memory. Instead, when the user presses Alt-P, we will want to allocate some memory, read the music in from a file, close the file, play the notes, free the memory, then go back to sleep.

It would be so nice if things worked this way, but they don't. The problem is that in this example we have no control over when my_int9_handler() will be invoked. Recall that my_int9_handler() is not called from within the program, the way functions like _dos_setvect() or go_tsr() are. Instead, my_int9_handler() is called whenever the user pounds on the keyboard: it is an asynchronous event that bears no relation to the internal state of whatever program happens to be running, or the internal state of DOS.

For instance, the foreground program might be copying a large file to the printer when the user decides that it's time for a musical interlude. While the foreground program is currently executing an INT 21h function such as Read File or Write File, the penzance() function suddenly takes over and starts issuing its own INT 21h requests. Will this work?

Not without a lot of help from us, it won't. The problem with this scenario is that MS-DOS is (quite rightly) designed as a single-task operating system, and that the code for INT 21h is not set up so it can be interrupted in the middle of

one DOS request, made to carry out some other DOS request, and allowed to resume where it was interrupted.

This property of MS-DOS is often referred to as *nonreentrancy*, meaning that, if INT 21h is already executing, another INT 21h request can't be issued. Reentrant code can be called by multiple processes simultaneously. It is designed so that one process in the function can be interrupted at any time, allowing another process to enter the function (hence the term *reentrant*). All variables are stored on the caller's stack, which is a unique aspect of the caller.

Nonreentrancy, it should be noted, is by no means a purely DOS issue. Any textbook on operating systems or on concurrent programming probably contains a discussion of the difference between reentrant code, which may be shared by several processes simultaneously, and what by contrast is called *serially reusable code*, which may be used by only one process at a time. MS-DOS contains serially reusable code.

When MS-DOS is called via INT 21h, it switches to one of three internal stacks: the I/O stack, the Disk stack, or the Auxiliary stack. Functions 00 through 0Ch use the I/O stack. The remainder of the functions use the Disk stack. If MS-DOS is called during a critical error (such as DIR A: when the drive door is open), the Auxiliary stack is used. Because of this stack-switching, if a TSR calls MS-DOS when the foreground is already executing inside INT 21h, MS-DOS will load the TSR's data onto its stack, overwriting the foreground process' data.

If DOS happens to be servicing a Function 0Ch request or lower, and our TSR issues a Function 0Dh request or higher, then there won't be a problem, because two different stacks are involved. Furthermore, a few INT 21h functions (33h, 50h, 51h, 62h, and 64h) are so simple that they use the caller's stack and are therefore fully reentrant. But for the most part, DOS is non-reentrant.

On the other hand, it's not just DOS we have to worry about. What if the heads on the hard disk are in the middle of writing data as part of the response to an application's INT 13h call? If our TSR starts issuing INT 13h requests that move the head somewhere else, then we're going to have a big reentrancy problem that has nothing to do with stacks or reusable code, but that could well result in a scrambled hard disk.

Does this mean our TSR *can't* perform DOS memory and file operations whenever the user presses the hotkey? Does all this have to be done once during initialization, before hooking any interrupt vectors, so that the memory-resident portion of our TSR avoids all use of DOS calls? For example, one book on C pro-

gramming for the PC makes the blanket statement that a TSR interrupt service routine (ISR) "cannot use any DOS functions." If this were true, it would certainly restrict what we could do with TSRs.

Actually, this isn't quite as terrible a restriction as it sounds. Many commercial programs for the PC that aren't even TSRs bypass DOS for many operations such as screen display and keyboard input. Avoiding DOS is not only possible, but, for certain key operations on the PC, it is practically a necessity. It is easy to write screen display functions, for example, that not only bypass DOS, but which are many times faster than DOS output routines, and which provide far greater control over the screen.

However, as we noted in chapter 4, one area of DOS functionality really is irreplaceable: file I/O. In addition, while programs can allocate expanded or extended memory rather than use the DOS memory allocation routines, expanded or extended memory is not always available, so many TSRs need to allocate memory via DOS. Basically, most TSRs need to make some INT 21h calls while popped up.

Fortunately, it is simply not true that TSR interrupt service routines can't make INT 21h calls. But it is true that TSRs must do something special in order to make such calls. There are two options:

- Defer issuing INT 21h calls while INT 21h is already in the middle of processing a request, or
- Somehow save and restore all of DOS's context (including the three DOS stacks) so that we *can* freely interrupt it.

The second option will be discussed later in this chapter, in the section on the "DOS Swappable Data Area" (SDA). Until then, the topic will be ways of not entering DOS in the middle of some other program's INT 21h call, but instead waiting until that call has completed: using DOS as a serially-reusable resource. Until then, you will be reading about the state that a TSR must save and restore as part of its popup regime.

The key requirement here is to have some way of determining when INT 21h is busy or, more accurately, of determining when one of its three stacks is in use. A short while ago, we saw a small block of pseudocode for trapping INT 21h calls, and it may have occurred to the reader that this might be used to determine whether DOS is being used. For example, we might put both our INT 21h han-

dler and INT 9 handler into the same program, and use the former to tell the latter whether it's safe to pop up:

```
INTERRUPT PTR old_int9_handler;      // keyboard
INTERRUPT PTR old_int21_handler;     // DOS
WORD using_io_stack = 0;
WORD using_disk_stack = 0;

INTERRUPT my_int21_handler()
    IF (ah <= 0x0c)
        INCR using_io_stack;
        CALL PTR old_int_21_handler;
        DECR using_io_stack;
    ELSE
        INCR using_disk_stack;
        CALL PTR old_int21_handler;
        DECR using_disk_stack;

INTERRUPT my_int9_handler()
    IF key == alt_m AND NOT using_disk_stack
        mikado();
    ELSE IF key == alt_p AND NOT using_disk_stack
        penzance();
    ELSE
        CALL PTR old_int9_handler();

BEGIN
    old_int9_handler = _dos_getvect(9);
    old_int21_handler = _dos_getvect(0x21);
    _dos_setvect(0x21, my_int21_handler);
    _dos_setvect(9, my_int9_handler);
    go_tsr();
```

We have hooked INT 21h so we can find out whether someone is "in DOS." The INT 21h handler increments a flag on entry to an INT 21h call, and decrements it on the way back out. The INT 9 handler checks the using_disk_stack flag, and won't pop up if it is non-zero. The flag therefore acts as a *semaphore*, serializing access to DOS.

This gives the basic idea behind making DOS calls from a TSR, but there are many problems with the preceding pseudocode. For example, DOS termination functions (Functions 00h, 31h, and 4Ch) do not return, and therefore would need to get special treatment. Likewise, this code does not take account of DOS critical errors. Nor does it account for the fact that a PC sitting at the COMMAND.COM

prompt is actually parked *inside* INT 21h Function 0Ah (Buffered Keyboard Input), making it seem as if one can't pop up while at the DOS prompt, which we know not to be the case.

Fortunately, we don't need to get this code to work properly, because MS-DOS *already* provides an INDOS semaphore and a "critical error" semaphore that our TSR can check. Instead of hooking INT 21h in an attempt to maintain our own INDOS flag, we can use the one that DOS already provides. (On the other hand, this technique of hooking an interrupt in order to maintain an in-use flag will be essential later on, when we need to serialize access to INT 13h, the ROM BIOS disk interrupt.)

This is where undocumented DOS enters the picture, because the DOS functions that return the addresses of the INDOS and critical-error semaphores are undocumented. Furthermore, DOS generates "idle" interrupts (INT 28h) while inside INT 21h Function 0Ah. As will be discussed later, these undocumented workarounds were originally created for use in Microsoft's TSRs such as PRINT.COM.

The fact that Microsoft's own TSRs use these functions should tell you that developers who want to create robust TSRs probably need to use them as well. It goes against common sense to assert that using undocumented features will make a program more rather than less stable, but who said that TSR programming was supposed to make sense? The techniques for writing correct TSRs may not be a model of software engineering at its finest, and some of the undocumented functions for TSR support have the feel of glorified afterthoughts rather than parts of a well thought-out interface, but you'll need them if you want your program to survive in the PC marketplace.

If you're writing a TSR, you have probably already bought into a host of compatibility problems, and frankly, using undocumented DOS is the least of them. The reason industry pundits have spoken of a "TSR crisis" is not because of undocumented DOS, but because of keyboard conflicts, problems associated with popping up over screens in graphics mode, memory usage conflicts, and the like. Undocumented DOS is actually one of the saner areas in TSR programming.

MS-DOS TSRs

How did PC programmers find out about the undocumented TSR functions? From examining Microsoft's own TSRs, of course.

TSRs have been a part of MS-DOS since its initial release in 1981. M. Steven Baker notes in his fine article on "Safe Memory-Resident Programming" (The Waite Group's *MS-DOS Papers*, 1988) that TSRs were even available within the 64KB confines of the earlier CP/M operating system, in programs like Smartkey, Uniform, and Unspol. The only TSR program to ship with DOS 1.x was MODE. PRINT, GRAPHICS, and ASSIGN were added in DOS 2.x.

PRINT is the only DOS utility program that multitasks. You can be running an application at the same time that PRINT is printing a file. Only one of the two programs is actually running at any given instant, but the illusion of simultaneous operation is maintained by switching between them on each timer tick.

When the PRINT program is installed, it chains into the BIOS timer-tick interrupt (INT 1Ch), the DOS Keyboard Busy Loop interrupt (INT 28h), and numerous other interrupts. Hooking a large number of interrupts is fairly normal for TSR operation. INT 1Ch and INT 28h allow the PRINT program to gain control at regular intervals, independent of the user, to perform its processing (open file, read, print and close). These intervals are sufficiently close together that your foreground program appears to be operating at the same time as the PRINT program.

When PRINT performs its processing, it saves the current DTA, PSP, and the vectors for INT 1Bh (Ctrl-Break), 23h (Ctrl-C), and 24h (Critical Error), and sets up its own values for these items. Upon exit from its current processing, it restores these values to their original state. The TSRs presented in this chapter follow a similar structure. The multitasking TSR at the end of this chapter is actually an enhancement to the PRINT utility: it will periodically look for files that appear in a certain subdirectory and automatically submit them to PRINT.

GRAPHICS allows graphics screens to be printed with the Shift-PrtSc key combination. It replaces the current INT 5 vector with a pointer to its own code. This code translates the data in the display adapter's memory into data recognized by the printer.

One of the significant improvements in MS-DOS 2.0 was the availability of a hard disk. However, this new disk (usually drive C:) did cause some problems with software that was hard-coded to use drive A: or B:. The ASSIGN utility allows drive letters to be mapped to other drive letters. When programs reference drive A:, they can be transparently made to instead access drive C:. ASSIGN sits on three MS-DOS interrupts: INT 21h (DOS Function Call), INT 25h (Absolute Sector Read), and INT 26h (Absolute Sector Write). When INT 25h or 26h is

called, and the AL register references the ASSIGNed drive, the value in AL is replaced by the new drive number. Simple, huh? MS-DOS does not translate calls to INT 13 (BIOS Disk Services), but it would be easy to write a utility for such a purpose.

MODE is a good example of a TSR that modifies output to a device. Many programs do not support a serial printer (COM1); they just reference LPT1. MODE grabs data sent via INT 17h (BIOS Parallel Printer Service), and sends the data to the serial port. The same principle can be used to write translation programs for various output devices. A TSR one of the authors wrote was for a printer that did not support the form feed command. The TSR sat on the parallel printer output interrupt and checked for a form-feed character. When one came by, the TSR would output the appropriate number of line feeds to get to the next page. It was a simple program, but it saved someone from having to buy a new printer.

The Generic TSR

Now it's time for some code! Our goal is to build a generic TSR with Microsoft C: this chapter provides everything so that, when a user-defined hotkey is pressed, a function named application() is called. You simply provide application(), link with the generic-TSR object modules, and you've got a TSR. We have deliberately stayed away from issues involving screen modes or even screen saves and restores, though, since these have nothing to do with undocumented DOS. The generic TSR code deals with all the issues connected with undocumented DOS. The resulting TSRs have been tested extensively, and seem quite robust (there are no 100% guarantees in the world of TSRs, however).

We will use our generic TSR to build three sample pop-ups: a simple file browser (TSRFILE), a memory-resident version of the MCB walker from chapter 3 (TSRMEM), and a memory-resident version of the INT 2Eh command interpreter from chapter 6 (TSR2E). We will also build MULTI, the non-pop-up multitasking program mentioned earlier. The TSRs can be built either using the "traditional" undocumented DOS functions for TSRs, or using the DOSSWAP technique described later on. Finally, you can indicate whether a given TSR uses the disk or not. In order to show how all these pieces fit together, we take the somewhat unusual approach of *first* showing the MAKEFILE for this project. The following file works with NMAKE from Microsoft C 6.0:

```
# NMAKE makefile for generic TSR
# example: C:\UNDOC>nmake tsrfile.exe

# can be overridden from environment with NMAKE /E
# example:
#   C:\UNDOC>set swap=1
#   C:\UNDOC>nmake /e tsrfile.exe
#
#   C:\UNDOC>set no_disk=1
#   C:\UNDOC>nmake /e tsrmem.exe
SWAP = 0
NO_DISK = 0

!IF $(SWAP)
DOSSWAP = -DDOS_SWAP
DOSSWAP_O = dosswap.obj
!ENDIF

!IF $(NO_DISK)
USES_DISK =
!ELSE
USES_DISK = -DUSES_DISK
!ENDIF

# defines the key components of the generic TSR:
#   TSREXAMP.C - main
#   INDOS.C - InDOS, critical error flag
#   PSP.C - Set PSP, Get PSP
#   EXTERR.C - Extended error save and restore
#   TSRUTIL.ASM - Miscellaneous routines
#   STACK.ASM - Stack save and restore
#   DOSSWAP.C - Optional use of DOS Swappable Data Area (SDA)
UNDOC_OBJS = indos.obj psp.obj exterr.obj

TSR_OBJS = tsrexamp.obj $(UNDOC_OBJS) $(DOSSWAP_O) \
    tsrutil.obj stack.obj

# command to turn a .C file into an .OBJ file
.c.obj:
    cl -AS -Ox -Zp -c -W3 -DTSR $(USES_DISK) $(DOSSWAP) $*.c

# command to turn an .ASM file into an .OBJ file
.asm.obj:
    masm -ml $*.asm;

# special handling for MULTUTIL.ASM
multutil.obj:  tsrutil.asm
```

```
    masm -ml -DMULTI tsrutil,multutil;

multstk.obj:  stack.asm
    masm -ml -DMULTI stack,multstk;

# make the file-browser sample TSR
tsrfile.exe:  $(TSR_OBJS) file.obj
    link /map/far/noi $(TSR_OBJS) file,tsrfile.exe,tsrfile.map;

# make the MCB-walker sample TSR
tsrmem.exe:  $(TSR_OBJS) mem2.obj put.obj
    link /far/noi $(TSR_OBJS) mem2 put,tsrmem.exe;

# make the INT 2Eh command-interpreter sample TSR
INT2E_OBJS = test2e.obj send2e.obj have2e.obj do2e.obj
INT2E = test2e send2e have2e do2e

tsr2e.exe: $(TSR_OBJS) $(INT2E_OBJS) put.obj
    link /far/noi $(TSR_OBJS) $(INT2E) put,tsr2e;

# make the non-pop-up PRINT add-on
multi.exe: multi.obj $(UNDOC_OBJS) multutil.obj multstk.obj
    link /far/noi multi $(UNDOC_OBJS) multutil multstk,multi;
```

TSR Programming in Microsoft C

Before actually beginning our examination of the various components of the generic TSR, there needs to be a discussion about writing TSRs in Microsoft C, rather than in assembly language. This discussion strays fairly far afield from the topic of undocumented DOS, unfortunately, but that's unavoidable: there are a lot of preliminaries to get out of the way before the discussion of TSR programming with undocumented DOS will make any sense. In any case, we will be reading about all sorts of interesting aspects of low-level PC programming in Microsoft C.

Managing TSRs in assembly language seems relatively easy at first because you have total control of the CPU. It becomes a bit more difficult when the actual TSR application goes beyond the scope of simple assembly-language code. C is generally easier to code than assembly language, and a wealth of libraries is available. By using C to write a TSR, you give up a little efficiency but gain ease of use and manageability.

In order for a TSR to do anything, it must be accessed via some type of interrupt. Therefore, anyone interested in TSR programming in a high level language like C must become familiar with the facilities for interrupt manipulation.

Most C compilers for the PC offer an interrupt or _interrupt keyword that can be used to create interrupt handlers (and thus, TSRs). The interrupt keyword causes the compiler to create special entry and exit code for any procedure whose definition has the interrupt modifier. On entry the function will save all of the registers and set DS to that of the C program. Because these registers are defined as parameters and are pushed on the stack, you can get and set them just like any other variable. When the procedure exits, it pops the registers values from the stack. An interrupt handler can be created like this:

```
typedef struct {
#ifdef __TURBOC__
    unsigned bp, di, si, ds, es, dx, cx, bx, ax;
#else
    unsigned es, ds;
    unsigned di, si, bp, sp, bx, dx, cx, ax;    /* PUSHA */
#endif
    unsigned ip, cs, flags;
    } INTERRUPT_REGS;

void interrupt far my_handler(INTERRUPT_REGS r)
{
    unsigned i = r.ax;
    r.bx = i >> 8;
}
```

Sample code for the interrupt keyword often shows an enormous parameter list for each interrupt handler, with each register named separately. Using the INTERRUPT_REGS structure (not a pointer to one!) makes the parameter list more manageable.

What sort of code does this really produce? By compiling with the Microsoft C -Fa or -Fc command-line switches, we can examine the resulting assembly-language code. The following is the code generated by the compilation of this interrupt procedure in Microsoft C. The order in which registers are pushed is dictated in part by the Intel PUSHA and POPA instructions. Note that Borland Turbo C also offers an interrupt keyword, but that the order in which registers are pushed and popped is different (and incompatible with the PUSHA/POPA instructions):

```
_my_handler PROC FAR
    push    ax              ; bp+18
    push    cx              ; bp+16
    push    dx              ; bp+14
    push    bx              ; bp+12
    push    sp              ; bp+10
    push    bp              ; bp+8
    push    si              ; bp+6
    push    di              ; bp+4
    push    ds              ; bp+2
    push    es              ; bp+0
    mov bp,sp
    sub sp,2
    mov ax,DGROUP
    mov ds,ax
    ASSUME DS:DGROUP
    cld
    mov ax,WORD PTR [bp+18] ; i = r.ax
    mov al,ah
    sub ah,ah
    mov WORD PTR [bp+12],ax ; r.bx = i >> 8
    mov sp,bp
    pop es
    pop ds
    pop di
    pop si
    pop bp
    pop bx
    pop bx
    pop dx
    pop cx
    pop ax
    iret
_my_handler PROC FAR
```

Pushing the registers on the stack allows the C function to access the register values through variables. Because these values are popped from the stack on exit, the C function can actually change the return values of registers on interrupt exit. Notice that on exit BX is popped twice. On entry, SP was pushed at this point. If the C function was allowed to change SP, the IRET instruction would put us in some unknown spot (recall that IRET uses the stack to return to the caller).

Note that CS:IP and the flags are pushed on the stack by the processor itself.

If we compile for 80286 and higher machines with -G2 switch, the resulting code now looks like this:

```
    .286
_my_handler PROC FAR
    pusha                       ; push ax,cx,dx,bx,old_sp,bp,si,di
    push ds
    push es
    mov bp,sp
    sub sp,2
    mov ax,DGROUP
    mov ds,ax
    ASSUME DS:DGROUP
    cld
    mov ax,WORD PTR [bp+18]
    mov al,ah
    sub ah,ah
    mov WORD PTR [bp+12],ax
    mov sp,bp
    pop es
    pop ds
    popa                        ; pop di,si,bp; skip sp; pop bx,dx,cx,ax
    iret
_my_handler ENDP
```

The enormous amount of code generated (even for the best case, with PUSHA/POPA) and the large amount of stack space used for our three-line interrupt handler should not go unnoticed. If you were writing my_handler in assembly language to begin with, it might look like this:

```
_my_handler PROC FAR
    mov bx, ax
    shr bx, 8
    iret
_my_handler ENDP
```

Every feature has a price, and here too we pay for the convenience of writing our application in C rather than in assembly language.

In addition to the interrupt or _interrupt keyword, C compilers for the PC generally offer a set of functions for manipulating interrupts. In Microsoft C, the DOS.H header file provides a large set of DOS-specific functions, including the following:

```
void (_cdecl _interrupt _far * _cdecl _dos_getvect(unsigned intno))();

void _cdecl _dos_setvect(unsigned intno,
    void (_cdecl _interrupt _far *new_handler)());

void _cdecl _chain_intr(void (_cdecl _interrupt _far *target)());
```

The functions _dos_getvect() and _dos_setvect() directly translate into calls to INT 21h Functions 25h and 35h, and are vastly preferable to using the more general intdosx() or int86x() functions. For example:

```
#include <dos.h>
// ...
extern void interrupt far my_int21_handler(); // declare new function
void (_interrupt _far *old_int21)();          // old function pointer

main()
{
    old_int21 = _dos_getvect(0x21);           // save old
    _dos_setvect(0x21, my_int21_handler);     // install new
    // ...
    _dos_setvect(0x21, old_int21);            // restore old
}
```

We can do other things with the old_int21 function pointer than restore it when we're finished, though. In fact, almost all interrupt handlers (and TSRs) *will* need to do something else with the pointer to the previous handler: chain to it! Microsoft C provides the extremely useful _chain_intr() function (unfortunately, Turbo C does not), which is necessary when your new interrupt handler needs to do preprocessing before chaining to the old handler. For instance:

```
void (_interrupt _far *old_int21)();

void interrupt far my_int21_handler(INTERRUPT_REGS r)
{
    // do some work
    _chain_intr(old_int21);
    // never reached!
}

main()
{
    old_int21 = _dos_getvect(0x21);           // save old
```

```
    _dos_setvect(0x21, my_int21_handler);      // install new
    // ...
}
```

If you need to do more work *after* chaining to the old handler, then you can't use _chain_intr(). Instead, you must directly call through the saved function pointer:

```
// maybe do some preprocessing
(*old_int21)();
// we're back: do postprocessing
```

The C compiler turns the call through the interrupt function pointer into the following:

```
pushf
call dword ptr old_int21
```

The problem with this, however, is that the compiler uses the CPU registers in ways that may not be obvious from an examination of your C code. The registers on entry to the old interrupt handler may therefore not be correct. This is not a problem with _chain_intr() because that function (which can only be correctly called from within a interrupt function) loads up the CPU registers with the "image" of the registers that was stored on the stack. Wherever possible, use _chain_intr(old) rather than (*old)().

There are various tradeoffs involved in writing interrupt handlers in C rather than in assembly language. All in all, it seems like a "win," but the overhead of pushing *all* registers on the stack on entry to an interrupt handler, and the inconvenience of not knowing the exact state of the registers before chaining to the previous handler, are sometimes too much. Fortunately, any time C is less convenient, we can always switch into assembly language: the generic TSR uses two assembly language modules, TSRUTIL.ASM and STACK.ASM, because that made more sense than any religious principles about only using C.

Keeping a Microsoft C Program Resident

The hardest thing to do from a TSR in C is estimate the amount of memory you actually want to keep resident. Microsoft C provides a handy _dos_keep() function (declared in DOS.H), which calls the DOS TSR function:

```
void _cdecl _dos_keep(unsigned retcode, unsigned memsize);
// retcode -- exit status code
// memsize -- allocated resident memory in 16-byte paragraphs
```

But this leaves unanswered the question of what number to pass in as memsize. When coding in assembly language, you can easily come up with this number, because you usually know the size of your code (and furthermore, have total control over its arrangement so you can jettison the startup code). In C, however, you do not control the memory structure of the program beyond your source code. The memory map for a small-model Microsoft C program, with hypothetical segment addresses, looks like this:

```
        ---------------
1E20h   FAR HEAP
        ---------------
        STACK
        NEAR HEAP
0E19h   DS
        ---------------
0BFAh   CODE
        ---------------
0BEAh   PSP
        ---------------
```

Of course, you could just pass a very high number to _dos_keep(), but with TSRs one of the primary goals is to keep memory consumption to the absolute minimum. The following code fragment details one way to keep a memory segment resident in C for small memory models:

```
#define PSP_ENV_ADDR    0x2c  /* environment address from PSP */
#define STACK_SIZE      8192  /* must be 16 byte boundary */

#define PARAGRAPHS(x)   ((FP_OFF(x) + 15) >> 4)

char far *stack_ptr;          /* pointer to TSR stack */
unsigned memtop;              /* number of paragraphs to keep */

// ...
/* MALLOC a stack for our TSR section */
stack_ptr = malloc(STACK_SIZE);
stack_ptr += STACK_SIZE;

//
```

```
/* release environment back to MS-DOS */
FP_SEG(fp) = _psp;
FP_OFF(fp) = PSP_ENV_ADDR;
_dos_freemem(*fp);

/* release unused heap to MS-DOS */
/* All mallocs for TSR section must be done in TSR init */
segread(&sregs);
memtop = sregs.ds + PARAGRAPHS(stack_ptr) - _psp;
_dos_setblock(memtop, _psp, &dummy);
_dos_keep(0, memtop);
```

First, create a block of memory in the near heap using malloc(). This will become the TSR's stack during activation. Instead of using this local stack, you can use whatever stack happens to be in effect during TSR activation. This is fine for small programs, but if you are doing wild and wonderful things with your code, it is best to create your own stack to avoid overflowing the foreground's stack. The stack size is added to the stack pointer variable to get it to the bottom of the stack. This stack bottom will become the top TSR in memory.

Using the value of the new stack pointer, the number of 16-byte paragraphs that must be kept resident (memtop) is calculated. The expression PARAGRAPHS(stack_ptr) gives us the number of paragraphs in our local heap. This number must be retained because it includes the mallocs we've already done. This is added to DS to establish the top of the memory we will need, and our PSP is subtracted to find the actual number of paragraphs needed by the entire program. A call is then issued to MS-DOS to shrink the current block down to the size specified. In simple programs created with the generic TSR, memtop was generally less than 600h paragraphs, giving the resulting TSR an in-memory footprint of about 24KB. This eliminates any far heap, the original C stack, and the unused near heap.

Using this method, you must perform any near mallocs before the creation of the stack. Once the TSR is resident, it cannot make any calls to the malloc family or use library routines that use malloc functions, because the near heap is gone. Use of malloc calls will vary with compiler implementation, so be sure to select your functions carefully. (The *Run-Time Library Reference* for Microsoft C 6.0 includes, in the entry for malloc, a list of all functions that call malloc: it's rather large.)

The final step is to call _dos_keep, which does an INT 21h Function 31h to terminate and stay resident, retaining in memory the number of paragraphs

specified. If all goes well, you should be able to find your TSR in the display from chapter 3's MEM program. For example:

```
C:\UNDOC>tsrfile -k 59 4
Activation: CTRL SCAN=59
C:\UNDOC>mem
Seg     Owner   Size
...
0BDA    1E76    000D (   208)
0BE8    0000    0000 (     0)   free
0BE9    0BEA    0597 ( 22896)            -k 59 4  [08 09 13 28 2F ]
...
```

The MEM display shows that the TSR begins at 0BEAh (its MCB, of course, is at 0BE9h), and that it retains 0597h paragraphs, or 22KB. The fact that we freed the environment is also reflected: we can still get the command line (which, by the way, designates a hotkey of Ctrl-F1), but the program name is no longer available. As for the various interrupts MEM says we've hooked, these will be discussed in a short while.

One final note about staying resident in C: in order to reduce their memory footprint even further, many TSRs jettison their startup code. For example, we don't need main() once we've gone resident: any subsequent calls to TSRFILE (to deinstall, for instance) are going to go to the main() of a completely different instance of the program, not to the main() of our resident copy. (We have one program, but possibly more than one process.) Anyhow, it would be nice to throw away the code for main(). Figuring out how to do this, given the memory map shown earlier, is left as an exercise for the reader. Don't stay up too late!

Not Going Resident

If you are writing your own TSR from scratch (rather than using a generic TSR such as the one we present here), it is a good idea to put off going resident for as long as possible. Don't try debugging your application as a TSR. Instead, have it spawn a command shell from which you can exit, or have it run a single program whose name and arguments appear on the DOS command line:

```
main(int argc, char *argv[])
{
    // TSR init goes here
    old_int09 = _dos_getvect(0x09);
```

```
    _dos_setvect(0x09, my_int09_handler);

#ifdef TESTING
    // to launch a command shell:
    system(getenv("COMSPEC"));

    // or, to run just one program:
    // spawnvp(P_WAIT, argv[1], &argv[1]);

    // we're back: do TSR deinstall
    _dos_setvect(0x09, old_int09);
#else
    // ...
    _dos_keep(0, memtop);
#endif
}
```

In fact, this is so handy you might consider making some of your applications into shells rather than TSRs. A program that needs to set up a "context" of some sort for another program is often best treated as a shell, not a TSR.

Jiggling the Stack

Remember the stack we created with malloc just before remaining resident? In order for that stack to be used, the TSR interrupt routine that performs activation must call two routines: one to set up the local stack on entry and one to restore the original stack on exit. The actual code to do the stack switch must be programmed in assembly language, because we don't have full access to the registers in C (though we could use in-line assembler in Microsoft C 6.0).

The following is a short assembly language module that manages the stack context switch. The _set_stack procedure saves the current foreground stack in our data area and sets the stack pointer to the stack created with malloc (stack_ptr). Notice the stack manipulation at entry and exit of this procedure. A return address was placed on the stack when set_stack was called. Because we are switching stacks, this address is popped from the stack on entry and pushed on the stack before exit. The _restore_stack procedure restores the stack segment and pointer to what was saved in _set_stack:

```
;STACK.ASM

;Define segment names used by C
```

```
;
_TEXT   segment byte public 'CODE'
_TEXT   ends

CONST   segment word public 'CONST'
CONST   ends

_BSS    segment word public 'BSS'
_BSS    ends

_DATA   segment word public 'DATA'
_DATA   ends

DGROUP  GROUP   CONST, _BSS, _DATA

    assume  CS:_TEXT, DS:DGROUP

    public  _set_stack, _restore_stack
    extrn   _stack_ptr:near     ;our TSR stack
    extrn   _ss_save:near       ;save foreground SS
    extrn   _sp_save:near       ;save foreground SP

_TEXT   segment
;*****
;void far set_stack(void) -
;   save current stack and setup our local stack
;*****
_set_stack  proc    far

;save foreground stack

;we need to get the return values from the stack
;since the current stack will change
    pop ax  ;get return offset
    pop bx  ;get return segment

;save away foreground process' stack
    mov word ptr _ss_save,ss
    mov word ptr _sp_save,sp

;setup our local stack
    mov ss,word ptr _stack_ptr+2
    mov sp,word ptr _stack_ptr

IFDEF MULTI
   mov bp,sp   ;make bp relative to our stack frame
ENDIF
```

```
;setup for ret
    push    bx
    push    ax

    ret
_set_stack  endp

;*****
;void far restore_stack(void) -
;   restore foreground stack, throw ours away
;*****
_restore_stack  proc    far

;we need to get the return values from the stack
;since the current stack will change
    pop cx  ;get return offset
    pop bx  ;get return segment

;save background stack
    mov word ptr _stack_ptr+2,ss
    mov word ptr _stack_ptr,sp

;restore foreground stack here
    mov ss,word ptr _ss_save
    mov sp,word ptr _sp_save

IFDEF MULTI
   mov bp,sp   ;make bp relative to our stack frame
ENDIF

;setup for ret
    push bx
    push cx

    ret
_restore_stack  endp
_TEXT   ends

_DATA   segment

_DATA   ends

    end
```

One final note on the stack: it's crucial that we compile with -Gs (or with a switch like -Ox that includes -Gs), to turn off stack checking. The C compiler's _chkstk routine would get hopelessly confused by the new stack we've created.

Undocumented DOS Functions for TSRs

Finally, we are ready to discuss TSR programming with undocumented DOS. Recall that the whole issue is how one makes DOS INT 21h calls from the resident portion of a TSR. First the traditional use of undocumented DOS will be presented, and afterward the new DOSSWAP technique will be shown.

MS-DOS Flags

DOS keeps a byte in memory called the INDOS flag, also known as the DOS safe flag. This flag indicates when it is safe to access MS-DOS functions, and is a semaphore that turns DOS into a serially reusable resource. The idea is that no one can enter DOS (make INT 21h calls) if the semaphore indicates that DOS is busy. Actually, as we'll see, there are numerous exceptions to this rule, but the basic idea is sound.

Generally, the activation section of a TSR that uses MS-DOS will check this flag. If it indicates that MS-DOS is busy, the TSR program must defer the activation, or at least that portion of the activation which actually makes INT 21h calls. Using undocumented INT 21h Function 34h, you can get the address of the INDOS flag. Because this address (returned in ES:BX) is constant for a particular operating environment, the initialization section of a TSR will call this function once. We can then store the address in a local variable for later access during the pop-up phase.

The following C module contains a function, DosBusy, that will return a zero if it is safe to make INT 21h calls. If DOS cannot be interrupted, it will return nonzero. Your application must call InitInDos (to set the addresses of the InDos flags) during initialization. If you neglect to do so, DosBusy will always return nonzero. (In a programming language like C++, you can make such initializations occur automatically.)

```
/* INDOS.C - Functions to manage DOS flags */

#include <stdlib.h>
#include <dos.h>

#define GET_INDOS        0x34
#define GET_CRIT_ERR     0x5D06

char far *indos_ptr=0;
char far *crit_err_ptr=0;
```

```
int   DosBusy(void);
void  InitInDos(void);

/*****
Function: Init InDos Pointers
Initialize pointers to InDos Flags
*****/
void InitInDos(void)
{
    union  REGS regs;
    struct SREGS segregs;

    regs.h.ah = GET_INDOS;
    intdosx(&regs,&regs,&segregs);
    /* pointer to flag is returned in ES:BX */
    FP_SEG(indos_ptr) = segregs.es;
    FP_OFF(indos_ptr) = regs.x.bx;

    if (_osmajor < 3)  /* flag is one byte after InDos */
        crit_err_ptr = indos_ptr + 1;
    else if (_osmajor==3 && _osminor == 0) /* flag is 1 byte before*/
        crit_err_ptr = indos_ptr - 1;
    else
    {
        regs.x.ax = GET_CRIT_ERR;
        intdosx(&regs,&regs,&segregs);
        /* pointer to flag is returned in DS:SI */
        FP_SEG(crit_err_ptr) = segregs.ds;
        FP_OFF(crit_err_ptr) = regs.x.si;
    }
}

/*****
Function: DosBusy
This function will non-zero if DOS is busy
*****/
int DosBusy(void)
{
    if (indos_ptr && crit_err_ptr)
        return (*crit_err_ptr || *indos_ptr);
    else
        return 0xFFFF;  /* return dos busy if flags are not set */
}

/*****
Function: Int28DosBusy
This function will return non-zero if the InDos flag is > 1 or
```

```
the critical error flag is non zero. To be used inside of an
INT 28 loop. Note that inside INT 28, InDOS == 1 is normal, and
indicates DOS is *not* busy; InDOS > 1 inside INT 28 means it is.
*****/
int Int28DosBusy(void)
{
    if (indos_ptr && crit_err_ptr)
      return (*crit_err_ptr || (*indos_ptr > 1));
    else
      return 0xFFFF;  /* return dos busy if flags are not set */
}
```

The preceding example puts the cart before the horse by referencing a byte in addition to the INDOS flag. This is the Critical Error flag. The Dos Critical Error flag is set when DOS is processing a critical error (of course!). It is yet another flag that must be checked before deciding if it is safe to access MS-DOS. In MS-DOS version 2.x, this flag is one byte after the INDOS flag. In MS-DOS version 3.x, this flag is one byte before the INDOS flag. In MS-DOS versions 3.10 and above, the address of the Critical Error flag can be retrieved by calling INT 21h Function 5D06h. It is a good idea to use this function, because the location of this flag may or may not always be dependent on the INDOS flag in future DOS releases. This function requires DOS version-checking. Be careful when you are writing your own functions that perform DOS version checks. Because of the extensive number of DOS levels, the procedure can be confusing.

Note that all the DOS versionitis problems are taken care of during initialization, in InitINDOS, so that DosBusy, which is called quite frequently, has an easy job.

In addition to checking if we are in the middle of a critical error, another use for the critical-error flag is to force MS-DOS to use its critical-error stack. A bug in MS-DOS 2.x requires that the critical-error flag be set (and therefore DOS's critical-error stack be used) so that the Get PSP and Set PSP functions (discussed momentarily) work properly. Crazy, huh? Being a PC programmer means taking a perverse pleasure in knowing odd facts like this.

As yet another forward reference, note that INDOS.C also provides a function called Int28DosBusy(), to be used in an INT 28h handler. Inside INT 28h, the INDOS flag will always be at least 1: in this context, (INDOS == 1) means DOS *isn't* busy (this is one of the many INDOS exceptions we were talking about), but if (INDOS > 1), then DOS is really busy: come back some other time!

Get/Set PSP

As discussed in chapter 3 (particularly the section "Unique Process Identifier"), each process in MS-DOS has a Program Segment Prefix (PSP). We saw that Memory Control Blocks (MCBs) are stamped with the PSP of their owner. In chapter 4, we saw that this 256-byte area contains, among other things, the default file handle table (Job File Table, JFT) for the process. Because it is a unique value (though this can get complicated in 80386 multitasking environments), the segment address of the PSP can also be used as a unique process identifier.

At any given moment in an MS-DOS system, there is a "current PSP." In the Appendix A entry for INT 21h Function 5D06, you can see that the current PSP is kept at offset 10h in the DOS Swappable Data Area (SDA), along with similar values such as the current Disk Transfer Area (DTA) and the current drive. When DOS received an INT 21h Function 3Dh request to open a file, for example, the handle returned in AX will be in a index into the JFT of the "current PSP."

Well, that's obviously the PSP that belongs to whatever process called INT 21h Function 3Dh, right?

No! Remember that we are writing TSRs here. The "current PSP," unless we somehow change it, belongs to whatever process happened to be running when we popped up. Thus, if our TSR decides to start opening files, it would be using the foreground process' PSP. This could be totally benign or totally disastrous.

Consider the following example of a TSR that leaves the "current PSP" alone when it pops up. If the TSR opens a file handle or allocates memory via MS-DOS, these items become associated with the foreground process. The foreground process is not aware of these items, but entries in its JFT will be used up. When the foreground terminates, all open files are closed and allocated memory segments are freed. This includes those which the TSR thought of as its own, yet allowed to be associated with the foreground process.

Really what our TSR should do when it pops up is somehow change DOS's current PSP so that it corresponds to the TSR, carry out whatever task our TSR is supposed to perform when it pops up, and then, before lapsing back into its dormant state, restore the current PSP to whatever value it had when we popped up.

Fortunately, DOS provides just the functions we need. Undocumented INT 21h Functions 50h and 51h are the Get PSP and Set PSP functions in MS-DOS 2.x and above. In DOS 3.x and above, documented Function 62h is also available to Get PSP. As noted in chapter 3, it is often thought that Get PSP returns the PSP of

whatever program called it. We now know that it gets DOS's "current PSP" out of the SDA. Likewise, Set PSP sets this value in the SDA.

In DOS 3.0 and higher, Functions 50h, 51h, and 62h do not use any of the DOS stacks and are fully reentrant: they are among the few INT 21h functions you can call without paying attention to the INDOS flag (and thereby, they constitute another exception to the "INDOS" rule). But, as noted earlier, to call Functions 50h or 51h in DOS 2.x, you must first force use of the critical-error stack.

The following pseudocode describes the steps to use Get/Set PSP from a TSR:

```
TSR_INITIALIZATION:
     .
     .
     psp_addr :=
     Get current PSP with Function 51h or 62h
     (this PSP will be that of the TSR)
     .
     Terminate and stay resident

TSR_ACTIVATION:
     fgrnd_psp_addr :=
     Get current PSP with Function 51h or 62h
     (since the TSR interrupted the foreground, this address
        will be that of the foreground process)

     Set current PSP with Function 50h, using psp_addr
     Do TSR work
     Set current PSP with Function 50h, using fgrnd_psp_addr
     Go back to sleep
```

The following C module, PSP.C, contains functions that Get and Set the current PSP, taking into account the various oddities which we have discussed: these are not just simple-minded sugar coating for the equivalent DOS functions. We test for the DOS version and set the critical-error flag accordingly. Again, you must call InitInDos before using the following functions:

```
/* PSP.C */

#include <stdlib.h>
#include <dos.h>
#include "tsr.h"

#define GET_PSP_DOS2   0x51
```

```
#define GET_PSP_DOS3   0x62
#define SET_PSP        0x50

static union  REGS regs;

/*****
Function: GetPSP - returns current PSP
*****/
unsigned GetPSP(void)
{
    if (_osmajor == 2)
    {
        if (! crit_err_ptr) /* must not have called InitInDos */
            return 0;

        *crit_err_ptr = 0xFF;   /* force use of proper stack */
        regs.h.ah = GET_PSP_DOS2;
        intdos(&regs,&regs);
        *crit_err_ptr = 0;
    }
    else
    {
        regs.h.ah = GET_PSP_DOS3;
        intdos(&regs,&regs);
    }

    return regs.x.bx;
}

/*****
Function: SetPSP - sets current PSP
*****/
void SetPSP(unsigned segPSP)
{
    if (!crit_err_ptr)          /* must not have called InitInDos */
        return;

    *crit_err_ptr = 0xFF;       /* force use of correct stack    */
    regs.h.ah = SET_PSP;
    regs.x.bx = segPSP;         /* pass segment value to set     */

    intdos(&regs,&regs);
    *crit_err_ptr = 0;
}
```

Extended Error Information

Consider the following scenario:

The foreground program has performed a DOS function that detected an error condition. Because of this, DOS has stored extended error information. Normally the foreground program can access the extended information at this point. If the TSR becomes active, however, the access of the extended error information will be delayed. Now the TSR has control and could possibly perform DOS functions that detect errors. This error detection will overwrite the existing extended error information, making it invalid for the interrupted (foreground) program.

Don't despair, though: DOS has this problem well in hand. In DOS 3.0 and higher, documented Function 59h is available to query the extended error information. A TSR must save this information prior to activation, and reset it at exit. Undocumented DOS Function 5D0Ah allows the extended error information to be set. On entry to 215D0A, DS:DX points to a table containing three words that represent the AX, BX, and CX registers returned by the call to 2159.

The following C functions can be used to get the extended error information on TSR activation and then to reset the extended error information on exit:

```
/* EXTERR.C - extended error saving and restoring */

#include <stdlib.h>
#include <dos.h>

#define GET_EXTERR      0x59
#define SET_EXTERR      0x5d0a

#pragma pack(1)

struct   ExtErr
{
   unsigned int errax;
   unsigned int errbx;
   unsigned int errcx;
};

void  GetExtErr(struct ExtErr * ErrInfo);
void  SetExtErr(struct ExtErr * ErrInfo);

/*****
Function: GetExtErr
get extended error information
```

```
*****/
void GetExtErr(struct ExtErr * ErrInfo)
{
   union  REGS regs;

   if (_osmajor >= 3)   /* only for DOS 3 and above */
   {
      regs.h.ah = GET_EXTERR;
      regs.x.bx = 0;  /* must be zero */
      intdos(&regs,&regs);
      ErrInfo->errax = regs.x.ax;
      ErrInfo->errbx = regs.x.bx;
      ErrInfo->errcx = regs.x.cx;
   }
}

/*****
Function: SetExtErr
set extended error information
*****/
void SetExtErr(struct ExtErr near * ErrInfo)
{
   union  REGS regs;
   struct SREGS segregs;

   if (_osmajor >= 3)   /* only for DOS  3 and above */
   {
      regs.x.ax = SET_EXTERR;
      regs.x.bx = 0;       /* must be zero */
      segread(&segregs);   /* put address of err info in DS:DX */
      regs.x.dx = (int) ErrInfo;
      intdosx(&regs,&regs,&segregs);
   }
}
```

Interrupt 28h

Our TSR can now pop up whenever DOS is not in use. We're sitting at the COMMAND.COM prompt, not doing anything, we hit the TSR's hotkey, and the TSR...Doesn't pop up!

This wasn't the answer you expected, was it? If we're not doing anything, the TSR should pop up as soon as we press its hotkey. Sitting at the COMMAND.COM seems like the epitome of idleness: why doesn't the TSR pop up?

The answer is quite simple: COMMAND is waiting for input *in DOS*. As explained in chapter 6, whenever COMMAND has finished carrying out some task

and awaits your next instruction, it calls the documented DOS Buffered Keyboard Input function (INT 21h Function 0Ah). This function provides the standard DOS editing keys such as F3. While idling at the prompt waiting for you to type something, COMMAND is parked inside INT 21h Function 0Ah. In other words, the INDOS flag is set.

Now what do we do? One alternative, naturally, is to throw in the towel and declare that our TSR won't pop up at the COMMAND prompt. We're probably not going to sell a lot of copies of the program that way, though.

This curious paradox—the INDOS flag is set and yet we know that DOS is really idle—must have confronted Microsoft when it was putting together the PRINT program. True, PRINT is not a pop-up, but the same principles apply: when COMMAND is "doing nothing" would seem to be a good time for the TSR to print some files. And, if you've ever used PRINT, you know that in fact it *does* print in the background while COMMAND is idling. So how did Microsoft resolve this dilemma?

They put in a hack so that, whenever DOS is waiting for a user keypress in places like Function 0Ah, it periodically generates an interrupt, INT 28h. PRINT hooks this interrupt, thereby receiving wakeup calls while the state of the INDOS flag otherwise indicates that it shouldn't. INT 28h is referred to as the MS-DOS Idle interrupt or Keyboard Busy Loop interrupt. Whenever this Idle interrupt is generated, it is safe to use INT 21h Functions 0Dh and above, as long as the INDOS flag is not greater than one. INT 28h is undocumented, but it is such a foundation of TSR programming that it is supported even in the DOS compatibility box of OS/2.

Thus, our TSR has acquired another wrinkle: in addition to checking the INDOS and critical-error flags, saving and restoring DOS's current PSP, and saving and restoring the extended error information, we now must hook INT 28h too. Well, no one ever said the DOS TSR interface was a model of clarity. The fact that the interface is entirely undocumented tells us that no one at Microsoft sat down and tried to design a nice interface for TSR programming. Instead, they put in what they needed to write their own TSRs. The end result is an interface that looks like something someone would design only for their own use. On the other hand, the DOS TSR interface (if we can call it that) benefits from the fact that its designers actually used it themselves. They ran into the same problems you run into with your TSRs, so they put in solutions.

Interrupt handlers for INT 28h should pass control to the previous INT 28h owner when complete. Generally they should not "hog" the INT 28h interrupt by executing large amounts of code. TSRs that solicit user input should not only hook INT 28h, but also periodically *invoke* INT 28h in their input loop. This gives other TSRs a chance to use the idle time. In our generic TSR, we use INT 28h to detect if the user had earlier pressed the hotkey at a time when we couldn't pop up. In the MULTI TSR program at the end of this chapter, INT 28h is hooked so that we can use the timeslices we get while the system is sitting at the COMMAND prompt.

The INDOS.C module shown earlier contains the function called Int28DosBusy(), which returns zero if it is safe to access DOS during an INT 28h. The INDOS flag will never be zero during an INT 28h, so you might think that we don't need to even check INDOS during an INT 28h. However, INDOS could be greater than one, in which case we *still* can't pop up.

The INT 28h handler appears in the main TSR module, TSREXAMP.C, to which we now turn.

Inside the Generic TSR

The main module for our generic TSR, TSREXAMP.C, includes initialization, the pop-up routine calls your application, and several interrupt handlers. The remainder of the interrupt handlers are written in assembly language (for reasons noted earlier) and are found in TSRUTIL.ASM. Naturally, TSREXAMP.C relies heavily on the modules we have already examined: INDOS.C, PSP.C, EXTERR.C, and STACK.ASM.

Rather than plunge directly into the 550 lines of code belonging to TSREXAMP.C, or the 250 lines that comprise TSRUTIL.ASM, it makes more sense to start off with a pseudocode explanation.

The following pseudocode makes heavy use of the keyword ON (borrowed from BASIC, which in turn borrowed it from PL/I). A phrase such as ON TIMER indicates code that is called, not from within the program itself, but from outside the program: it is merely an interrupt handler, written in C using the interrupt keyword discussed earlier, and installed using (in this example) _dos_setvect(8, new_int8), but as we will see the ON keyword as used in the following psuedocode is particularly expressive of what happens in our TSR.

Note that the following discussion assumes that the TSR is going to access the disk during its pop-up phase, and also that the TSR does *not* use the DOSSWAP interface (which was mentioned but not really discussed in detail).

The initialization of our generic TSR looks something like this:

```
INIT ; main() in TSREXAMP.C
    IF they want to deinstall
        CALL deinstall()
    ELSE IF TSR not already installed
        MALLOC stack
        GETVECT TIMER(8), KEY(9), DISK(13h), IDLE(28h), MULTIPLEX(2Fh)
        ; MULTIPLEX (2Fh) for communication with already-resident
        ; (install check, deinstall)
        SETVECT TIMER, KEY, DISK, IDLE, MULTIPLEX
        RELEASE environment
        RELEASE unused heap
        TSR
```

There are no surprises here, except perhaps the fact that we are somehow using INT 2Fh to communicate between an already-resident copy of the TSR program and a second copy that, rather than go TSR, is used simply to deinstall the resident copy. If TSRFILE has already been made resident, it can be deinstalled by typing TSRFILE -d at the COMMAND prompt. If we are not deinstalling, then the TSR first checks to see if the TSR is already installed (also using INT 2Fh, incidentally). If it isn't, when the INIT routine completes, our program has become memory resident, and five interrupt handlers have been installed.

Before examining the interrupt handlers, let us create a few semaphores that the interrupt handlers will use to communicate among themselves:

```
FLAG wanted_pop up        ; wanted to pop up earlier, but DOS was busy
FLAG disk_unsafe         ; INT 13h in use?
FLAG idle_int            ; is INT 28h in progress?
```

Note that the INDOS flag is not included here, because this flag is maintained by DOS itself, and is not located inside our program; we use our DosBusy() routine to check the INDOS flag.

The first interrupt handler we need to examine is that one that handles keyboard events. Because we installed an INT 9 handler, each time the user presses a key, any key, a piece of code something like the following will get executed:

```
ON KEY ; new_int9() in TSREXAMP.C
    IF we are not already running AND
        IF it's our hotkey AND
            IF NOT disk_unsafe THEN
                CALL POP UP
            ELSE
                ; we can't pop up now, so just set flag indicating
                ; that we WANT to pop up at next available moment
                wanted_pop up = TRUE
                CALL previous KEY handler
        ELSE
            ; not our hotkey - chain to next handler
            CALL previous KEY handler
    ELSE
        ; we're already running - let key be processed normally
        CALL previous KEY handler
```

In the simplest scenario, the user presses the hotkey at a time when INT 13h isn't in use. Our keyboard handler then calls the POP UP routine:

```
POP UP ; tsr_function() in TSREXAMP.C
    CALL set_stack() ; switch to our own stack
    IF DosBusy() AND NOT idle_int
        wanted_pop up = TRUE
    ELSE
        ; we really can POP UP now!
        GETVECT CTRL-BREAK(1Bh), CTRL-C(1Ch), CRITERR(24h)
        SETVECT CTRL-BREAK, CTRL-C, CRITERR
        current_PSP = GetPSP()
        CALL SetPSP(TSR_PSP)
        current_DTA = GetDTA()
        CALL SetDTA(TSR_DTA)
        save_err = GetExtErr()
        eat keys
        CALL application()
        CALL SetExtErr(save_err)
        CALL SetDTA(current_DTA)
        CALL SetPSP(current_PSP)
        SETVECT CTRL-BREAK, CTRL-C, CRITERR ; REVERT

ON CTRL-BREAK DO NOTHING

ON CTRL-C DO NOTHING

ON CRITERR ; new_int24() in TSREXAMP.C
    RETURN FAILURE
```

Continuing with the simplest scenario, let's say that DosBusy() returns FALSE. We then proceed to install three short-term interrupt handlers. The Ctrl-Break and Ctrl-C handlers merely discard these events: a more sophisticated TSR might do something fancy with them. The Critical Error handler merely returns failure. The key point is that the pop-up portion of our TSR must run its own handlers for these events, not whatever handler the foreground process happens to have installed at the time. Next, we swap our own Disk Transfer Area (DTA) and PSP with that of the foreground process, and save the extended error information discussed earlier. We eat whatever keys are lurking in the keyboard buffer and—finally!—call the application (which, as we know, actually does something useful like providing a notepad, dialing a modem, or playing a tune from *Pinafore*). When the application is finished, we put everything back the way we found it.

That was the *simplest* scenario. Say the user has pressed the hotkey, but we can't pop up: either INT 13h is in use, or DOS is "really busy" (that is, the INDOS flag is set *and* we're not inside an INT 28h idle interrupt). In this case, either the keyboard handler or the pop up routine sets the wanted_popup flag, and more or less immediately returns (in the case of the keyboard handler, with an IRET).

So all we've done is set the wanted_popup flag: how is this going to actually help us pop up?

Remember the INT 8 timer tick handler we installed? At each timer tick (about 18.2 times a second, unless someone has reprogrammed the chip that generates these interrupts), our TIMER routine gets woken up. Its job is to check the wanted_popup flag:

```
ON TIMER ; new_int8() in TSREXAMP.C
    CALL previous TIMER handler
    IF NOT tsr_active
        IF wanted_popup
            IF NOT DosBusy()
                IF NOT disk_unsafe
                    wanted_popup = FALSE
                    CALL POP UP
```

Once the wanted_popup flag has been set, the TIMER routine will, 18.2 times a second, see if it's *now* safe to pop up. It will do this until it actually is safe to pop up, at which time the flag is turned off.

One thing not shown in pseudocode, but appearing in the genuine code in TSREXAMP.C and TSRUTIL.ASM, is that, for all hardware interrupts like INT 8 or INT 9, we chain to the previous handler. In addition to giving the previous interrupt handler an opportunity to do its thing, we also rely on the previous handler to send the end-of-interrupt (EOI) command to the Intel 8259A interrupt controller. This is why you won't find the otherwise-obligatory out(0x20,0x20) sprinkled throughout the code.

In addition to timer ticks, we can also use the Idle interrupt as a trigger for servicing a wanted_popup request. Note also that the IDLE handler increments and decrements the idle_int flag which is checked on entry to the POP UP routine:

```
ON IDLE ; new_int28() in TSREXAMP.C
    INCR idle_int
    IF wanted_popup
        IF NOT Int28DosBusy()
            IF NOT tsr_active
                IF NOT disk_unsafe
                    POP UP!
    DECR idle_int
    CALL previous IDLE handler
```

Finally, how does the important disk_unsafe flag stay updated? There is unfortunately not a flag in the BIOS that we can get a far pointer to as we did with INDOS, so we hook INT 13h in order to create our own disk_unsafe semaphore:

```
ON DISK ; new_int13() in TSRUTIL.ASM
    INCR disk_unsafe
    CALL previous DISK handler
    DECR disk_unsafe
```

That's about all there is to our generic TSR. Note how decentralized the code for a TSR is: rather than have one top-level routine that calls various subroutines, we instead have a collection of independent handlers that will get called due to some event taking place outside the program. The system has no top, and instead consists of these asynchronously-invoked agents. Much is made of event-driven programming in environments like Windows, the Macintosh, or OS/2 Presentation Manager, but in reality it's not much different from what we're doing here.

Having taken this walk through the psuedocode, you should now be able to fully understand the actual live C source code in TSREXAMP.C. Many of the variable and function names are different from our psuedocode, though. Also, the sections which are conditionally compiled with #ifdef DOSSWAP haven't been explained yet:

```
/*
TSREXAMP.C
by Raymond J. Michels
with revisions by Tim Paterson
and Andrew Schulman
*/

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <bios.h>
#include "tsr.h"

#define STACK_SIZE      8192  /* must be 16 byte boundary */
#define SET_DTA         0x1a  /* SET Disk Transfer Address */
#define GET_DTA         0x2f  /* GET Disk Transfer Address */

#define DOS_EXIT        0x4C  /* DOS terminate (exit) */

#define KEYBOARD_PORT   0x60  /* KEYBOARD Data Port */

#define PSP_TERMINATE   0x0A  /* Termination addr. in our PSP */
#define PSP_PARENT_PSP  0x16  /* Parent's PSP from our PSP */
#define PSP_ENV_ADDR    0x2c  /* environment address from PSP */

#define HOT_KEY         32    /* Hot key along with ALT (D)*/

#define RIGHT_SHIFT     1
#define LEFT_SHIFT      2
#define CTRL_KEY        4
#define ALT_KEY         8

#define MULTIPLEX_ID    0xC0
#define INSTALL_CHECK   0x00
#define INSTALLED       0xFF
#define DEINSTALL       0x01
```

```
#define PARAGRAPHS(x)   ((FP_OFF(x) + 15) >> 4)

unsigned char multiplex_id = MULTIPLEX_ID;
char far *stack_ptr;            /* pointer to TSR stack */
unsigned ss_save;               /* slot for stack segment register */
unsigned sp_save;               /* slot for stack pointer register */
int tsr_already_active = 0;     /* true if TSR active */
int pop up_while_dos_busy = 0;  /* true if hotkey hit while dos busy */
int int_28_in_progress = 0;     /* true if INT 28 in progress */
int unsafe_flag = 0;            /* true if INT 13 in progress */
unsigned keycode;
unsigned foreground_psp;        /* PSP of process we've interrupted */
unsigned foreground_dta_seg;    /* DTA of process we've interrupted */
unsigned foreground_dta_off;
char buf[20];                   /* work buffer */
unsigned long TerminateAddr;    /* used during de-install */
union REGS regs;                /* register work structures */
struct SREGS sregs;
struct ExtErr ErrInfo;          /* save area for extended error info */
int hot_key;                    /* keycode for activation */
int shift_key;                  /* shift status bits (alt,  ctrl..) */
int user_key_set = 0;

/* Save areas for old interrupt pointers */
INTVECT old_int8, old_int9, old_int10, old_int13, old_int1b, old_int23;
INTVECT old_int24, old_int28, old_int2f;

#ifdef DOS_SWAP
extern int dos_critical;     /* used by DOSSWAP.C */
INTVECT old_int2a;
void interrupt far new_int2a(INTERRUPT_REGS);
#endif

/* PROTOTYPES FOR THIS MODULE */
void interrupt far new_int8(INTERRUPT_REGS);
void interrupt far new_int9(INTERRUPT_REGS);
extern void interrupt far new_int13(void);  /* in TSRUTIL.ASM */
void interrupt far new_int1b(INTERRUPT_REGS);
void interrupt far new_int23(INTERRUPT_REGS);
void interrupt far new_int24(INTERRUPT_REGS);
void interrupt far new_int28(INTERRUPT_REGS);
void interrupt far new_int2f(INTERRUPT_REGS);
void tsr_function(void);
void tsr_exit(void);
void usage(char *);
int UnlinkVect(int Vect, INTVECT NewInt, INTVECT OldInt);
void parse_cmd_line(int argc, char *argv[]);
```

```
void main(int argc,char *argv[]);

/*********
* TIMER INTERRUPT HANDLER
*********/
void interrupt far new_int8(INTERRUPT_REGS r)
{
    (*old_int8)();     /* process timer tic */

#ifdef DOS_SWAP
    if (!tsr_already_active && pop up_while_dos_busy &&
        !dos_critical && !unsafe_flag)
#else
    if (!tsr_already_active && pop up_while_dos_busy &&
        !DosBusy() && !unsafe_flag)
#endif
    {
        pop up_while_dos_busy = 0;
        tsr_already_active = 1;
         _enable(); /* turn interrupts back on */
         tsr_function();
         tsr_already_active = 0;
    }
}

/**********
* KEYBOARD INTERRUPT HANDLER
**********/
void interrupt far new_int9(INTERRUPT_REGS r)
{
    if (! tsr_already_active)
    {
         if ((keycode = inp(KEYBOARD_PORT)) != hot_key)
            _chain_intr(old_int9);

         if ((_bios_keybrd(_KEYBRD_SHIFTSTATUS) &
            shift_key) == shift_key)
         {
#ifdef USES_DISK
            if (!unsafe_flag)
            {
#endif
                popup_while_dos_busy = 0;
                tsr_already_active = 1;
                (*old_int9)();     /* send key to old int routine */
                tsr_function();
                tsr_already_active = 0;
```

```
#ifdef USES_DISK
            }
           else
            {
           popup_while_dos_busy = 1;
           _chain_intr (old_int 9);
#endif
         }
         else
            _chain_intr(old_int9);
     }
     else
         _chain_intr(old_int9);
}
/*********
* CTRL-BREAK INTERRUPT HANDLER
*********/
void interrupt far new_int1b(INTERRUPT_REGS r)
{
    /* do nothing */
}
/**********
* CTRL-C INTERRUPT HANDLER
**********/
void interrupt far new_int23(INTERRUPT_REGS r)
{
    /* do nothing */
}

/**********
* CRTITICAL ERROR INTERRUPT HANDLER
**********/
void interrupt far new_int24(INTERRUPT_REGS r)
{
    if (_osmajor >= 3)
        r.ax = 3;   /* fail dos function */
    else
        r.ax = 0;
}

/**********
* DOS IDLE INTERRUPT HANDLER
**********/
void interrupt far new_int28(INTERRUPT_REGS r)
{
    int_28_in_progress++;
```

```
#ifdef DOS_SWAP
    if (pop up_while_dos_busy && !dos_critical
        && !tsr_already_active && !unsafe_flag)
#else
    if (pop up_while_dos_busy && (!Int28DosBusy())
        && !tsr_already_active && !unsafe_flag)
#endif
    {
         tsr_already_active = 1;
         tsr_function();
         tsr_already_active = 0;
    }

    int_28_in_progress--;
    _chain_intr(old_int28);
}

#ifdef DOS_SWAP
/*********
* DOS INTERNAL INTERRUPT HANDLER
*********/
void interrupt far new_int2a(INTERRUPT_REGS r)
{
   switch (r.ax & 0xff00)
   {
      case 0x8000:    /* start critical section */
         dos_critical++;
         break;
      case 0x8100:    /* end critical section */
      case 0x8200:    /* end critical section */
         if (dos_critical)    /* don't go negative */
            dos_critical--;
         break;
      default:
         break;
    }
   _chain_intr(old_int2a);
}
#endif

/*********
* DOS MULTIPLEX INTERRUPT HANDLER
*********/
void interrupt far new_int2f(INTERRUPT_REGS r)
{
    unsigned ah = r.ax >> 8;
    unsigned al = r.ax & 0xFF;
```

```
    if (ah == multiplex_id)
    {
        if (al == INSTALL_CHECK)
            r.ax |= INSTALLED;
        else if (al == DEINSTALL)
        {
            // because of stack swap, pass arg in static variable.
             TerminateAddr = ((long)r.bx << 16) + r.dx;
             if (! tsr_already_active)  /* don't exit if we're active */
             {
                 _enable(); /* STI */
                 tsr_exit();
                 // If we got here, we weren't able to unlink
                 r.ax = 0xFFFF;  //let caller know we're still there
                 // MSC 6.0 /Ox optimizes the above instruction away
                 // get it back by using the value in ax
                 tsr_already_active = -r.ax;
                 // set to 1 to prevent any more action
             }
         }
    }
    else
         _chain_intr(old_int2f);
}

/**********
* TSR ACTIVE SECTION
**********/
void tsr_function()
{
    set_stack();

#ifdef DOS_SWAP
    if (SaveDosSwap() && !int_28_in_progress)
#else
    if (DosBusy() && !int_28_in_progress)
#endif
        pop up_while_dos_busy = 1; /* set flag: next INT 8,28 activates
us */
    else
    {
        pop up_while_dos_busy = 0;

        /* save old interrupt-CTRL-BREAK, CTRL-C and CRIT ERROR */
        old_int1b = _dos_getvect(0x1b);
        old_int23 = _dos_getvect(0x23);
        old_int24 = _dos_getvect(0x24);
```

```
        /* set our interrupts functions */
       _dos_setvect(0x1b, new_int1b);
       _dos_setvect(0x23, new_int23);
       _dos_setvect(0x24, new_int24);

        /* save current PSP and set to ours */
        /* not needed for DOSSWAP, but can be used by application */
        foreground_psp = GetPSP();

        SetPSP(_psp);   // _psp in STDLIB.H

#ifndef DOS_SWAP
        /* get foreground DTA */
        regs.h.ah = GET_DTA;
        intdosx(&regs, &regs, &sregs);
        foreground_dta_seg = sregs.es;
        foreground_dta_off = regs.x.bx;
#endif

        /* set up our DTA */
        regs.h.ah = SET_DTA;
        regs.x.dx = 0x80;   /* use default in PSP area */
        sregs.ds = _psp;
        intdosx(&regs, &regs, &sregs);

#ifndef DOS_SWAP
        /* Get Extended Error Information */
        GetExtErr(&ErrInfo);
#endif

        /* suck up key(s) in buffer */
        while (_bios_keybrd(_KEYBRD_READY))
           _bios_keybrd(_KEYBRD_READ);

        /* your code goes here */
        application();

#ifdef DOS_SWAP
        RestoreDosSwap();
#else
        /* put back extended error information */
        SetExtErr(&ErrInfo);

        /* put back original DTA */
        regs.h.ah = SET_DTA;
        regs.x.dx = foreground_dta_off;
        sregs.ds  = foreground_dta_seg;
```

```
        intdosx(&regs, &regs, &sregs);

        /* put back original PSP */
        SetPSP(foreground_psp);
#endif

        /* put back original INTS */
       _dos_setvect(0x1b, old_int1b);
       _dos_setvect(0x23, old_int23);
       _dos_setvect(0x24, old_int24);
    }

    restore_stack();
}

// only restores OldInt if someone hasn't grabbed away Vect
int UnlinkVect(int Vect, INTVECT NewInt, INTVECT OldInt)
{
    if (NewInt == _dos_getvect(Vect))
    {
        _dos_setvect(Vect, OldInt);
        return 0;
    }
    return 1;
}

void tsr_exit(void)
{
    set_stack();
    /* put interrupts back the way they were, if possible */

    if (!(UnlinkVect(8, new_int8, old_int8)      |
          UnlinkVect(9, new_int9, old_int9)      | // Do not use ||, we
          UnlinkVect(0x28, new_int28, old_int28) | // don't want early out
          UnlinkVect(0x13, new_int13, old_int13) |
#ifdef DOS_SWAP
          UnlinkVect(0x2a, new_int2a, old_int2a) |
#endif
          UnlinkVect(0x2f, new_int2f, old_int2f) ))
    {
        // Set parent PSP, stored in our own PSP, to the current PSP
        *(int far *)(((long)_psp << 16) + PSP_PARENT_PSP) = GetPSP();

        // Set terminate address in our PSP
        *(long far *)(((long)_psp << 16) + PSP_TERMINATE) =
TerminateAddr;
```

```
        /* set psp to be ours */
        SetPSP(_psp);

        /* exit program */
        bdos(DOS_EXIT, 0, 0);
    }
    restore_stack();
}

void usage(char *progname)
{
    fputs("Usage: ", stdout);
    puts(progname);
    puts(" [-d to deinstall] [-k keycode shift-status] [-f multiplex id]");
    puts(" Valid multiplex id");
    puts("    00 through 15 specifies a unique INT 2F ID");
    puts(" Valid shift-status is any combination of:");
    puts("    1 = Right Shift");
    puts("    2 = Left Shift");
    puts("    4 = CTRL");
    puts("    8 = ALT");
    exit(1);
}

void do_deinstall(char *progname)
{
    fputs(progname, stdout);
    switch (deinstall())
    {
        case 1:
            puts(" was not installed");
            break;
        case 2:
            puts(" deinstalled");
            break;
        default:
            puts(" deactivated but not removed");
            break;
    }
    exit(0);
}

int set_shift_key(unsigned sh)
{
    /* figure out, report on shift statuses */
    /* make sure shift key < 0x10 and non-zero */
```

```
    if (((shift_key = sh) < 0x10) && shift_key)
    {
        printf("Activation: %s%s%s%sSCAN=%d\n",
        shift_key & RIGHT_SHIFT ? "RIGHT " : "",
        shift_key & LEFT_SHIFT ? "LEFT " : "",
        shift_key & CTRL_KEY ? "CTRL " : "",
        shift_key & ALT_KEY ? "ALT " : "",
          hot_key);
        return 1;
    }
    else /* error, bad param */
    {
      puts("Invalid Shift-Status");
      return 0;
    }
}

void parse_cmd_line(int argc, char *argv[])
{
    int i;
    int tmp;

    for (i = 1; i < argc; i++)   /* for each cmdline arg */
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
            switch(toupper(argv[i][1]))
            {
                case 'D':
                    do_deinstall(argv[0]);
                    break;
                case 'K':    /* set pop up key sequence */
                    user_key_set = 1;
                    i++;     /* bump to next argument */
                    if ((hot_key = atoi(argv[i])) != 0)
                    {
                        i++;   /* bump to next argument */
                        if (! set_shift_key(atoi(argv[i])))
                            usage(argv[0]);
                    }
                    else
                        usage(argv[0]);
                    break;
                case 'F':   /* set multiplex ID */
                    i++;    /* bump to next argument */
                    if ((tmp = atoi(argv[i])) < 0x10)

                        multiplex_id += tmp; /* range of C0-CF */
                    else
```

```
                        usage(argv[0]);
                    break;
                default:    /* invalid argument */
                    usage(argv[0]);
            }   /* end switch */
        else
            usage(argv[0]);
}

void main(int argc,char *argv[])
{
    union REGS  regs;
    struct SREGS sregs;
    unsigned far *fp;
    unsigned memtop;
    unsigned dummy;

    InitInDos();

    parse_cmd_line(argc,argv);

    /* check if TSR already installed! */
    regs.h.ah = multiplex_id;
    regs.h.al = INSTALL_CHECK;
    int86(0x2f, &regs, &regs);
    if (regs.h.al == INSTALLED)
    {
        puts("TSR already installed");
        fputs(argv[0], stdout); puts(" -D de-installs");
        exit(1);
    }

    if (! user_key_set)
    {
        puts("Press ALT-D to activate TSR ");
        printf("Multiplex ID = %0x \n",multiplex_id);
        hot_key = HOT_KEY;
        shift_key = ALT_KEY;
    }

#ifdef DOS_SWAP
    if (InitDosSwap() != 0)
    {
        puts("Error initializing DOS Swappable Data Area");
        exit(1);
    }
#endif
```

```
    /* MALLOC a stack for our TSR section */
    stack_ptr = malloc(STACK_SIZE);
    stack_ptr += STACK_SIZE;

    /* get interrupt vector */
    old_int8  = _dos_getvect(8);      /* timer interrupt */
    old_int9  = _dos_getvect(9);      /* keyboard interrupt */
    old_int13 = _dos_getvect(0x13);   /* disk intr, in TSRUTIL.ASM */
    old_int28 = _dos_getvect(0x28);   /* dos idle */
    old_int2f = _dos_getvect(0x2f);   /* multiplex int */

#ifdef DOS_SWAP
    old_int2a = _dos_getvect(0x2a);   /* dos internal int */
#endif

    init_intr();  /* initialize int routines in TSRUTIL.ASM */

    /* set interrupts to our routines */
    _dos_setvect(8, new_int8);
    _dos_setvect(9, new_int9);
    _dos_setvect(0x13, new_int13);      /* in TSRUTIL.ASM */
    _dos_setvect(0x28, new_int28);
    _dos_setvect(0x2f, new_int2f);

#ifdef DOS_SWAP
    _dos_setvect(0x2a, new_int2a);
#endif

    /* release environment back to MS-DOS */
    FP_SEG(fp) = _psp;
    FP_OFF(fp) = PSP_ENV_ADDR;
    _dos_freemem(*fp);

    /* release unused heap to MS-DOS */
    /* All MALLOCS for TSR section must be done in TSR_INIT() */
    /* calculate top of memory, shrink block, and go TSR */
    segread(&sregs);
    memtop = sregs.ds + PARAGRAPHS(stack_ptr) - _psp;

    _dos_setblock(memtop, _psp, &dummy);
    _dos_keep(0, memtop);
}
```

TSREXAMP.C #includes the rather uninteresting, but necessary, TSR.H, which contains typedefs and function prototypes for all the modules that make up the generic TSR:

```
/* TSR Prototype file and common variables */

#define INTERRUPT void interrupt far

typedef struct {
    unsigned es, ds, di, si, bp, sp;
    unsigned bx, dx, cx, ax, ip, cs, flags;
    } INTERRUPT_REGS;

typedef void (interrupt far *INTVECT)();

/* Prototypes for functions in INDOS.C */
int   DosBusy(void);
int   Int28DosBusy(void);
void  InitInDos(void);

/* Prototypes for functions in PSP.C */
unsigned GetPSP(void);
void  SetPSP(unsigned segPSP);

/* Prototypes for functions in TSRUTIL.ASM */
int   far   deinstall(void);
void  far   init_intr(void);
void  far   idle_int_chain(void);
void  far   init_intr(void);

void  interrupt far   new_int10(void);
void  interrupt far   new_int13(void);
void  interrupt far   new_int25(void);
void  interrupt far   new_int26(void);

void  far   timer_int_chain(void);

/* Prototypes for functions in STACK.ASM */
void  far   set_stack(void);
void  far   restore_stack(void);

/* Prototypes for functions in EXTERR.C */
void  GetExtErr(struct ExtErr * ErrInfo);
void  SetExtErr(struct ExtErr * ErrInfo);

struct   ExtErr
{
   unsigned int errax;
   unsigned int errbx;
   unsigned int errcx;
};
```

```
/* Prototypes for functions in DOSSWAP.C */
int   InitDosSwap(void);
int   SaveDosSwap(void);
void  RestoreDosSwap(void);

/* Pointer defined in INDOS.C */
extern char far *   indos_ptr;
extern char far *   crit_err_ptr;
```

Finally, there's TSRUTIL.ASM, which contains miscellaneous routines which we either didn't want to write in C, or couldn't:

```
;TSRUTIL.ASM

;Define segment names used by C
;
_TEXT   segment byte public 'CODE'
_TEXT   ends

CONST   segment word public 'CONST'
CONST   ends

_BSS    segment word public 'BSS'
_BSS    ends

_DATA   segment word public 'DATA'
_DATA   ends

DGROUP  GROUP   CONST, _BSS, _DATA

    assume  CS:_TEXT, DS:DGROUP

    public  _new_int13, _init_intr
IFDEF MULTI
    public  _timer_int_chain
    public  _new_int10, _new_int25, _new_int26
ELSE
    public  _deinstall
ENDIF

    extrn   _ss_save:near       ;save foreground SS
    extrn   _sp_save:near       ;save foreground SP
    extrn   _unsafe_flag:near   ;if true, don't interrupt
    extrn   _old_int13:near
IFDEF MULTI
    extrn   _old_int8:near
```

```
    extrn   _old_int10:near
    extrn   _old_int25:near     ; note difference between
    extrn   _old_int26:near     ; old_int25 and _old_int25!
ELSE
    extrn   _multiplex_id:near  ;our int 2f id byte
ENDIF

_TEXT   segment

IFNDEF MULTI
;*****
;void far deinstall(void)
;function to use int 2f to ask TSR to deinstall itself
;the registers are probably all changed when our tsr exits
;so we save then and perform the INT 2f. The TSR exit will
;eventually bring us back here. Then the registers are restored
;This function is called from the foreground, not the TSR

DEINSTALL   equ     1

_deinstall       proc    far
    push si
    push di
    push bp
    mov word ptr _ss_save,ss         ;save our stack frame
    mov word ptr _sp_save,sp

    mov cs:_ds_save,ds      ; save DS for later restore

    mov bx,cs
    mov dx,offset TerminateAddr ;bx:dx points to terminate address
    mov ah, byte ptr _multiplex_id
    mov al, DEINSTALL
    int 2fh                 ;call our TSR
;
;if TSR terminates ok, we'll skip this code and return to Terminate Addr
;
    jmp short NoTerminate

TerminateAddr:
;Restore DS and stack
        mov     ax,cs:_ds_save  ;bring back our data segment
        mov     ds,ax           ;destroyed by int 2f

        mov al,2            ;Set value for success

        mov     ss, word ptr _ss_save   ;restore our stack
```

```
        mov     sp, word ptr _sp_save   ;destroyed by int 2f

NoTerminate:
        cbw             ;Extend return value to word
        pop bp
        pop di
        pop si
        ret
_deinstall endp
ENDIF

;*****
;void inc_unsafe_flag(void) - increment unsafe flag
;*****
inc_unsafe_flag proc    far
    push    ax
    push    ds
    mov ax,DGROUP       ;make DS = to our TSR C data segment
    mov ds,ax

    inc word ptr _unsafe_flag

    pop ds              ;put DS back to whatever it was
    pop ax
    ret
inc_unsafe_flag endp

;*****
;void dec_unsafe_flag(void) - decrement unsafe flag
;*****
dec_unsafe_flag proc    far
    push    ax
    push    ds
    mov ax,DGROUP       ;make DS = to our TSR 'C' data segment
    mov ds,ax

    dec word ptr _unsafe_flag

    pop ds              ;put DS back to whatever it was
    pop ax
    ret
dec_unsafe_flag endp

;we can't trap the following interrupts in C for a number of
;reasons
;   INT 13 returns info in the FLAGS, but a normal IRET
;   restores the flags
```

```
;
;   INT 25 & 26 leave the flags on the stack. The user
;   must pop the off after performing an INT 25 or 26
;
;   These interrupts pass information via registers such
;   as DS. We don't want to change DS.
;
;   Since DS is unknown, we must call the old interrupts
;   via variables in the code segment. The _init_intr routine
;   sets up these CS variables from ones with nearly-identical
;   names in the C data segment in TSREXAMP.C.

;*****
;void far init_intr(void)
;move interrupt pointer saved in the C program to our CS data area
;*****
_init_intr proc far
        push    es
        push    bx

IFDEF MULTI
        les     bx,dword ptr _old_int10
        mov     word ptr cs:old_int10,bx
        mov     word ptr cs:old_int10+2,es

        les     bx,dword ptr _old_int25
        mov     word ptr cs:old_int25,bx
        mov     word ptr cs:old_int25+2,es

        les     bx,dword ptr _old_int26
        mov     word ptr cs:old_int26,bx
        mov     word ptr cs:old_int26+2,es
ENDIF
        ; note incredibly confusing distinction
        ; between e.g. _old_int13 and old_int13
        les     bx,dword ptr _old_int13
        mov     word ptr cs:old_int13,bx
        mov     word ptr cs:old_int13+2,es

        pop     bx
        pop     es
        ret
_init_intr endp

;*****
;void far new_int13(void) - disk interrupt
;*****
```

```
_new_int13 proc far
    call    inc_unsafe_flag
    pushf   ;simulate interrupt call
    call    cs:old_int13
    call    dec_unsafe_flag
    ret 2   ; leave flags intact
_new_int13 endp

IFDEF MULTI
;*****
;void far new_int10(void) - video interrupt
;*****
_new_int10 proc far
    call    inc_unsafe_flag
    pushf   ;simulate interrupt call
    call    cs:old_int10
    call    dec_unsafe_flag
    iret
_new_int10 endp

;*****
;void far new_int25(void) - MS-DOS absolute sector read
;*****
_new_int25  proc far
    call    inc_unsafe_flag
    call    cs:old_int25
    call    dec_unsafe_flag
    ret ; user must pop flags - MS-DOS convention
        ; so leave them on the stack
_new_int25 endp

;*****
;void far new_int26(void) - MS-DOS absolute sector write
;*****
_new_int26 proc far
    call    inc_unsafe_flag
    call    cs:old_int26
    call    dec_unsafe_flag
    ret ; user must pop flags - MS-DOS convention
        ; so leave them on the stack
_new_int26 endp

;*****
;void far timer_int_chain(void) - jump to next timer ISR
;we need to clean up the stack because of this call
;*****
_timer_int_chain proc far
```

```
    mov _ax_save,ax
    pop ax
    pop ax
    mov ax,_ax_save
    jmp dword ptr _old_int8
_timer_int_chain    endp
ENDIF

;
;save areas for original interrupt vectors
;
IFDEF MULTI
old_int10   dd  0       ;video
old_int25   dd  0       ;sector read
old_int26   dd  0       ;sector write
ENDIF
old_int13   dd  0       ;disk

_ds_save    dw  0
_TEXT   ends

_DATA   segment
IFDEF MULTI
_ax_save dw 0
ENDIF
_DATA   ends

    end
```

TSR Command-Line Arguments

Any program built with the generic TSR can take optional command-line arguments that set its hotkey and its Multiplex Interrupt ID number. A command-line option is also available to *deinstall* the TSR, the implementation of which will be discussed in detail later on.

The command-line syntax is:

```
[tsrname] [-k scan shift] [-f multiplex_id] [-d deinstalls]
```

Valid shift-status is any combination of:

```
1 = Right Shift
2 = Left Shift
4 = CTRL
8 = ALT
```

```
Valid multiplex id

    00 through 15 specifies a unique INT 2F ID starting at AH=C0h
```

The default hotkey is Alt-D, and the default Multiplex ID is 0 (which turns into INT 2Fh Function C0h). Specifying alternate hotkeys and Multiplex IDs makes it possible to simultaneously run multiple programs built with the generic TSR. Finally, to deinstall a TSR whose Multiplex ID is not the default, you must specify use both the -F and -D switches. For example:

```
C:\UNDOC>tsrfile -k 59 8
    TSRFILE hotkey is Alt-F1 (F1 decimal scancode is 59)
    TSRFILE multiplex ID is default C0h

C:\UNDOC>tsrmem -k 60 4 -f 1
    TSRMEM hotkey is Ctrl-F2 (F2 decimal scancode is 60)
    TSRMEM multiplex ID is C1h

C:\UNDOC>tsrmem -f 1 -d
    TSRMEM deinstalled
```

Writing TSRs with the DOS Swappable Data Area (SDA)

Undocumented INT 21h Function 5D06h for DOS 3.1 through 3.3, and Function 5D0Bh for DOS 4.x, gives us access to the DOS SDA. This is a block of data that contains the current context of MS-DOS. The context of MS-DOS includes the current PSP segment, and the three MS-DOS stacks, as discussed earlier, and as shown in gory detail in Appendix A. If the List of Lists (LoL) is the key to DOS's data, then the SDA *is* DOS's data. For example, when INT 21h Function 51h or 62h return the current PSP, where do you think they get it from? From the SDA. Ever wonder *exactly* how large each of DOS's three stacks is? Just look at SDA: they're in there! In the previous chapter, we made extensive use of the SDA in order to implement our network redirector. The SDA is essentially the DOS data segment. In DOS 4 and higher, there can be multiple SDAs.

What does this have to do with TSRs? Rather than *wait* for some time when there's no danger of reentering DOS as we've been doing up to now, a TSR can actually use Functions 5D06h and 5D0Bh to safely *reenter* MS-DOS by saving and restoring the SDA. This allows us to call MS-DOS at almost any time without having to wait until the DOS flags indicate it is safe.

These functions are used in conjunction with INT 2Ah. When undocumented INT 2Ah is invoked, it indicates that DOS is in a "critical section." When DOS is in a critical section, you cannot change the SDA. The end of a critical section is indicated by a call to undocumented INT 2Ah Functions 81h or 82h. Note that INT 2Ah is invoked by MS-DOS and *not* by your application: you need to write an interrupt handler for INT 2Ah.

INT 21h Function 5D06h returns the following information:

```
DS:SI - points to DOS swappable data area
DX    - size of area to swap when InDOS > 0
CX    - size of area to always swap
```

INT 21h Function 5D0Bh (for DOS 4 and higher) returns in DS:SI a pointer to an SDA *list*, which contains:

```
Offset     Size       Description
------     ----       -----------
00h        WORD       Count of SDAs

SDA_ENTRY:
02h        DWORD      Address of this SDA

06h        WORD       Data area length and type:
                      bit 15    - set if swap always
                                  clear if swap while InDOS > 0
                      bits 14-0 - length in bytes

08h                   next SDA_ENTRY
```

To reiterate, these functions give us pointers to the data area(s) that contains all of the information related to the current process *and* information specific to a DOS call that may be in progress. Since MS-DOS switches stacks when invoked via INT 21h, it is seemingly not reentrant. But since the stacks are part of this data area, we can save their current information (by moving the entire swappable data area) and immediately call DOS without fear of trashing DOS's internal stacks and variables!

In previous sections, we checked the DOS flags to determine if it was safe to activate and, once activated, we saved the current PSP, DTA and extended error information. Using the DOSSWAP method you eliminate these steps. We can determine if it is safe to pop up by tracking the INT 2Ah critical-section calls. If we

are not in critical section, we can call DOS. If INDOS is zero, we just need to save the data area that is always swapped (typically 18h bytes). If INDOS is non-zero, then all swappable data areas must be saved (typically 73Ch bytes, less than 2KB).

We save the data to a memory block that was allocated during TSR initialization (see the InitDosSwap function in DOSSWAP.C below). In our C TSR, this malloc must be performed before the TSR stack is allocated. Once the data area has been saved, we set the current PSP and DTA values to those for our TSR. When it is time for the TSR to exit, we just move back the SDAs that we've saved. Since this data block contains the current PSP, DTA and extended error information, we don't need to deal directly with these values.

Note: At the time of this writing, is it not clear if the DOS SDA list returned by 215D0B is static when MS-DOS is booted, or if it is changed dynamically during the course of MS-DOS execution. It appears that 215D06 and 215D0B return identical information for MS-DOS 4.1. To keep room for future DOS changes, you may want to malloc a large block during TSR initialization to hold potentially many DOS data blocks.

The following C module contains functions for saving and restoring the DOS SDA:

```
/* DOSSWAP.C - Functions to manage DOS swap areas */

#include <stdlib.h>
#include <dos.h>
#include <memory.h>
#include "tsr.h"

#define GET_DOSSWAP3        0x5d06
#define GET_DOSSWAP4        0x5d0b

#define SWAP_LIST_LIMIT     20

struct  swap_list     /* format of DOS 4+ SDA list */
{
    void    far*    swap_ptr;
            int     swap_size;
};

/* variables for 3.x swap work */
static   char  far *  swap_ptr;  /* pointer to dos swap area */
static   char  far *  swap_save; /* pointer to our local save area */
```

```
static   int   swap_size_indos;
static   int   swap_size_always;
static   int   size;

/* variables for 4.x swap work */
static int swap_count;      /* count of swappable areas */
static struct swap_list swp_list[SWAP_LIST_LIMIT]; /*list of swap
areas*/
static char far *swp_save[SWAP_LIST_LIMIT];   /* out save area */
static int swp_flag[SWAP_LIST_LIMIT];   /* flags if has been swapped */

static int dos_level;    /* for level dependent code */
int dos_critical;       /* in critical section, can't swap */

/*****
Function: InitDosSwap
Initialize pointers and sizes of DOS swap area. Return zero if success
*****/
int InitDosSwap(void)
{
    union  REGS regs;
    struct SREGS segregs;

    if ((_osmajor == 3) && (_osminor >= 10))
        dos_level = 3;
    else if (_osmajor >= 4)
        dos_level = 4;
    else
        dos_level = 0;

    if (dos_level == 3)    /* use 215D06 */
    {
        regs.x.ax = GET_DOSSWAP3;
        intdosx(&regs,&regs,&segregs);
        /* pointer to swap area is returned in DS:SI */
        FP_SEG(swap_ptr) = segregs.ds;
        FP_OFF(swap_ptr) = regs.x.si;

        swap_size_indos = regs.x.cx;
        swap_size_always= regs.x.dx;

        size = 0;  /* initialize for later */
        return ((swap_save = malloc(swap_size_indos)) == 0);
    }
    else if (dos_level >= 4)  /* use 5d0b */
    {
        struct swap_list far *ptr;
```

```
        int far *iptr;
        int i;
        regs.x.ax = GET_DOSSWAP4;
        intdosx(&regs,&regs,&segregs);
        /* pointer to swap list is returned in DS:SI */
        FP_SEG(iptr) = segregs.ds;
        FP_OFF(iptr) = regs.x.si;
        swap_count = *iptr;                    /* get size of list */
        iptr++;
        ptr = (struct swap_list far *) iptr;  /* create point to list */

        if (swap_count > SWAP_LIST_LIMIT)      /* too many data areas */
            return 2;

        /* get pointers and sizes of data areas */
        for (i = 0; i < swap_count; i++)
        {
            swp_list[i].swap_ptr = ptr->swap_ptr;
            swp_list[i].swap_size= ptr->swap_size;
            if (!(swp_save[i] = malloc(swp_list[i].swap_size & 0x7fff)))
                return 3;   /* out of memory */
            swp_flag[i] = 0;
            ptr++;   /* point to next entry in the list */
        }
        return 0;
    }
    else
        return 1;   /* unsupported DOS */
}

/*****
Function: SaveDosSwap
This function will save the dos swap area to a local buffer
It returns zero on success, non-zero meaning can't swap
*****/
int SaveDosSwap(void)
{
    if (dos_level == 3)
    {
        if (swap_ptr && !dos_critical)
        {
            /* if INDOS flag is zero, use smaller swap size */
            size = (*indos_ptr) ? swap_size_indos : swap_size_always;

            movedata(FP_SEG(swap_ptr),  FP_OFF(swap_ptr),
                     FP_SEG(swap_save), FP_OFF(swap_save),
                     size);
```

```
        }
        else        /* can't swap it */
            return 1;
    }
    else if (dos_level == 4)
    {
        /* loop through pointer list and swap appropriate items */
        int i;
        for (i = 0; i < swap_count; i++)
        {
            if (swp_list[i].swap_size & 0x8000)  /* swap always */
            {
                movedata(FP_SEG(swp_list[i].swap_ptr),
                         FP_OFF(swp_list[i].swap_ptr),
                         FP_SEG(swp_save[i]),
                         FP_OFF(swp_save[i]),
                         swp_list[i].swap_size & 0x7fff);
            }
            else if (*indos_ptr)     /* swap only if dos busy */
            {
                movedata(FP_SEG(swp_list[i].swap_ptr),
                         FP_OFF(swp_list[i].swap_ptr),
                         FP_SEG(swp_save[i]),
                         FP_OFF(swp_save[i]),
                         swp_list[i].swap_size);
            }
        }
    }
    else
        return 1;

    return 0;
}

/*****
Function: RestoreDosSwap
This function will restore a previously swapped dos data area
*****/
void RestoreDosSwap(void)
{
    if (dos_level == 3)
    {
        /* make sure its already saved and we have a good ptr */
        if (size && swap_ptr)
        {
            movedata(FP_SEG(swap_save), FP_OFF(swap_save),
                     FP_SEG(swap_ptr),  FP_OFF(swap_ptr), size);
```

```
            size = 0;
        }
    }
    else if (dos_level == 4)
    {
        int i;
        for (i = 0; i < swap_count; i++)
        {
            movedata(FP_SEG(swp_save[i]),
                     FP_OFF(swp_save[i]),
                     FP_SEG(swp_list[i].swap_ptr),
                     FP_OFF(swp_list[i].swap_ptr),
                     swp_list[i].swap_size);
           swp_flag[i] = 0;   /* clear flag */
        }
    }
}
```

To try out the new DOSSWAP method for building TSRs, recompile TSREXAMP.C with -DDOSSWAP, and link with the DOSSWAP module: see the makefile shown earlier in this chapter.

To use the DOSSWAP method in the multitasking non-pop-up example presented later, we would need to save not only the foreground data area (data belonging to the process we are interrupting), but the background data area (our TSR's data) as well. The SDA for the TSR could be saved during TSR initialization. Using this method, we would not need to deal with PSP, DTA, and Extended Error values at all since they would already exist in the SDA! By always saving and restoring the data area, it may make it easier to design some sort of round-robin task switcher. A hotkey could step through a number of independent applications. Interestingly, the Microsoft Windows 3.0 multitasker uses the undocumented SDA functions.

If you have examined the actual contents of the DOS SDA in our appendix, you can see that this area is quite large. Because of this you may need to weigh the advantages and disadvantages of using the SDA. The primary advantage of using the SDA in TSRs is that you can activate almost anytime while DOS is busy (unless, of course, a critical section has been flagged via INT 2Ah). This would be most beneficial for multitasking or round-robin task switching, since the response to a task switch would be almost instantaneous. In our generic pop-up TSR, for example, using DOSSWAP allows us to pop up instantly in the middle of a TYPE command.

There are two disadvantages. One is that this function requires memory to save the data area(s). This problem could be lessened by swapping the data to extended or expanded memory. Second, this technique is new and its effectiveness has yet to be determined. It's something that you must play with, and prove to yourself that it really works.

Removing a TSR

A TSR can often, but not always, be safely and completely removed from memory. When a TSR is removed correctly, there will be no evidence that it was ever installed. Recall our discussion in chapter 4 of orphaned file handles in the System File Table (SFT). The final proof of total removal is demonstrated if the TSR is initially loaded with output redirection; for example, TSRFILE >TSROUT. The redirected output file is closed and the handle released only if the TSR is fully removed in the proper way. Otherwise, you may release memory and put back interrupt vectors, but there's still that orphan languishing in the SFT. The FREEUP utility presented in chapter 4 was just a work around. Here, you will see how to remove a TSR so you don't leave behind orphans in the first place.

There is one condition that will always prevent the full removal of a TSR. If, at the time TSR removal is attempted, a subsequent program has chained into any of the same interrupt vectors as our TSR, then our TSR must not be removed. The problem is that there is no way to unlink our TSR from the middle of the interrupt chain.

Consider the case of TSRFILE, when some other TSR has loaded afterward and chained into INT 9, the keyboard interrupt. When a key is typed, this other TSR will be called first, since its address is now the one stored in the interrupt vector table. It may or may not process the keystroke, but in any case it will call the routine whose address was in the interrupt table before it took over—that will be TSRFILE. If TSRFILE is no longer in memory, the system will crash.

We would probably like to be able to stop the new TSR from calling TSRFILE, and instead have it call the address that was there before TSRFILE loaded—the one stored in old_int9. But there's no way to tell the new TSR to do that, or to find the place where it stores the chaining address so we can change it. And we can't just set the interrupt table back to the vectors we originally found there, because then the new TSR won't be chained in anymore. Not only will this keep the new TSR from working correctly, but it might crash the system if the new TSR is chained into some vectors (ones TSRFILE didn't use) but not others.

Another obstacle to fully removing a TSR is that the TSR and the current process must cooperate to make it work. This means that the TSR can be removed by typing a command, but it can't completely remove itself on its own. For example, a print spooler TSR cannot completely remove itself when the last job is done printing; nor can a pop-up TSR fully remove itself when an appropriate hot key is typed. In these cases it is possible to shrink memory usage to practically nothing, but the acid test (closing the redirected output file) will fail, leaving a file handle that is either permanently lost or that needs the FREEUP treatment.

Our generic TSR is removed from the command line. For example, TSRFILE -D. In other words, after one copy of TSRFILE has been installed as a TSR, a second copy can be loaded with the -D argument to remove the first.

The generic TSR uses INT 2Fh, the Multiplex Interrupt, for communication between these two copies of the program. The INT 2Fh function number (AH value) can be set by the user from the command line, but the subfunctions (AL value) are 0 (install check) and 1 (deinstall check). Note that the use of subfunction 0 for the install check is dictated by the standard Multiplex Interrupt interface.

Here are the required steps for deinstalling:

1. See if the interrupt vectors we chained into have been changed. Restore all that haven't. If any have, disable TSR operation but skip the remaining steps, leaving the TSR in memory.
2. The PSP of the TSR holds the value of its parent's PSP at offset 16h. Set this field to the PSP of the current process (i.e., the second non-resident copy of the TSR program).
3. The PSP of the TSR also holds the far address to return to when termination is complete, at offset 0Ah. Set this field to an appropriate address within the current process.
4. Set the current PSP to the PSP of the TSR.
5. Execute the normal DOS Terminate function (INT 21h Function 4Ch). This will free all memory allocated to the TSR, close all files, and use the parent-PSP and termination-address information placed in the PSP.
6. Execution resumes back in the originating process at the address set into the TSR's PSP. The PSP is set to the current process (because we put it in the TSR's PSP). All registers have unknown values, including the stack. SS and SP are set to the values they had when the TSR was invoked, which is a valid

stack but not within the current process's memory space. (Normally it is COMMAND.COM's stack.)

The assembly-language helper used to call the TSR with INT 2Fh restores the necessary registers (see _deinstall in TSRUTIL.ASM, shown earlier).

The first step, restoring the interrupt vectors, is always done by the TSR. The remaining steps can be done by the current process, if the TSR returns the value of its PSP via a prearranged interface (most likely, INT 2Fh).

However, the deinstall code in TSRUTIL.ASM and TSREXAMP.C works differently: it has the TSR do all the steps up to and including step 5, performing the DOS Terminate function. It starts when the second copy of the TSR, invoked with the -D option, performs an INT 2Fh with AX=C001h. This interrupt is intercepted by the TSR copy of TSRFILE, which assumes the terminate address is BX:DX. If any of the interrupt vectors used by the TSR have been chained by someone else, then the TSR returns from the INT 2Fh with AL=0FFh. Otherwise, the vectors are restored, and the terminate address kept in the PSP are set to the values passed in. The parent's PSP value, also kept in the PSP, is set by calling GetPSP (INT 21H Function 51h or 62h) to get the PSP of the current process.

It's possible that the INT 2Fh will instead return, indicating that one or more of the interrupt vectors have been changed and the TSR is unable to unlink, or that the TSR was not installed in the first place. In the first case, the TSR will have set AL=0FFh to let the caller know it tried to unlink but failed. In the second case, AL will be unchanged at 0. The generic TSR reports both of these conditions.

Sample TSR Programs

To exercise our generic TSR, we built three different simple TSRs, two of which are pop-up versions of programs from other parts of this book.

TSRFILE

The first sample program, TSRFILE, demonstrates that we really can make DOS file I/O and memory allocation calls while we're popped up in the middle of some other program. When TSRFILE pops up, it prompts the user for a filename and then displays the file on the screen. No screen saving/restoring amenities are provided.

FILE.C uses the Microsoft C _dos functions to do file I/O and memory allocation. These functions translate directly into the appropriate INT 21h calls, and are

thus preferable to using the intdos or int86 functions. Instructions for turning FILE.C into TSRFILE.EXE are found in the makefile shown earlier in this chapter; it can also be compiled as a stand-alone FILE.EXE:

```
#include <dos.h>
#include <conio.h>
#include <fcntl.h>
#include <share.h>

char file_prompt[] = "File? ";
char cant_open[] = "Can't open file\r\n";
char error_reading[] = "Error reading file\r\n";
char insuff_mem[] = "Insufficient memory; Press any key...\r\n";
char crlf[] = "\r\n";

#define PUTSTR(s)   \
    _dos_write(STDERR, (char far *) s, sizeof(s)-1, &wcount)

#define MIN_PARAS   4
#define WANT_PARAS  64
#define BYTES       (paras << 4)

#define STDERR      2

#ifdef TSR
application(void)
#else
main(void)
#endif
{
    char buf[81];
    char far *s;
    unsigned rcount, wcount, ret, paras, seg;
    int f;

    /* prompt for filename */
    if (PUTSTR(file_prompt) != 0)
        return;

    /* get filename */
    if ((_dos_read(STDERR, buf, 80, &rcount) != 0) || (rcount < 3))
        return;
    /* replace CRLF with NULL */
    buf[rcount-2] = '\0';

    /* try to allocate: first try a lot, then a little */
```

```
    if (_dos_allocmem(WANT_PARAS, &seg) == 0)
        paras = WANT_PARAS;
    else if (_dos_allocmem(MIN_PARAS, &seg) == 0)
        paras = MIN_PARAS;
    else
    {
        PUTSTR(insuff_mem);
        return;
    }
    FP_SEG(s) = seg;
    FP_OFF(s) = 0;

    /* open file */
    if (_dos_open(buf, O_RDWR | SH_DENYNO, &f) != 0)
        return PUTSTR(cant_open);

    /* display file */
    while (((ret = _dos_read(f, s, BYTES, &rcount)) == 0) && rcount)
        if (_dos_write(STDERR, s, rcount, &wcount) != 0)
            break;
    /* write one more CRLF */
    PUTSTR(crlf);
    if (ret)
        PUTSTR(error_reading);

    /* free memory */
    _dos_freemem(seg);

    /* close file */
    _dos_close(f);

    PUTSTR("Press any key...");
}
```

Note that FILE.C makes two stabs at allocating memory, because if we pop up over COMMAND.COM, there is almost no free memory available: the largest block in memory is used by COMMAND, and all that's left are little dribs and drabs (like the environment we freed during TSR initialization). We'll can see this situation when we run the MEM program as a TSR:

TSRMEM

One problem with the MEM program presented in chapter 3 was that, since it was a stand-alone program, we could only examine the memory map from within MEM itself. By putting the generic TSR and MEM together to form

TSRMEM.EXE, we can examine the memory map within other programs. For example, we can clearly see how COMMAND grabs the largest chunk of memory, leaving almost nothing free:

```
C:\UNDOC>tsrfile
C:\UNDOC>tsrmem -k 59 8 -f 1
C:\UNDOC>\sidekick\sk

[Hit TSRMEM hotkey, Alt-F1]
Seg     Owner   Size
09F3    0008    00F4 (  3904)   config [15 4B 67 ]
0AE8    0AE9    00D3 (  3376)   0BC1    c:\dos33\command.com  [22 2E ]
0BBC    0000    0003 (    48)   free
0BC0    0AE9    0019 (   400)
0BDA    171A    000D (   208)
0BE8    0000    0000 (     0)   free
0BE9    0BEA    0575 ( 22352)            [F1 FA ]
115F    1160    05B9 ( 23440)            -k 59 8 -f 1 [1B 23 24 2F F4
F5 ]
1719    171A    19B5 (105296)   0BDB    C:\SIDEKICK\SK.COM  [08 09 10 13
                                             16 1C 21 25 26 28 ]
30CF    0AE9    8730 (553728)           [30 F8 ]
B800
```

Note that the largest block in the MCB chain, totalling 8730h paragraphs, is not marked "free." Instead, it's owned by PSP 0AE9. Looking back along the MCB chain (which also functions as a PSP chain), we see that 0AE9 is COMMAND.COM. In fact, there are only three paragraphs of free memory, located directly after COMMAND. Even the environment we freed in TSRMEM was picked up by SideKick for use as *its* environment. If we hit TSRFILE's hotkey (Alt-D) at this point, it will ask us for the filename and then report "Insufficient memory."

However, if we leave COMMAND by running an application, and hit Alt-D, there will generally be plenty of memory, because the largest block is no longer being hogged by COMMAND. Again, this shows up clearly if we hit TSRMEM's hotkey within some other application (like Lugaru's Epsilon, used to edit this file). The MCB belonging to COMMAND.COM has now been replaced by the following:

```
30CF    30DE    000D (   208)
30DD    30DE    292D (168656)   30D0    c:\eps\EPSILON.EXE  [00 05 16 ]
5A0B    0000    5DF4 (384832)   free    [30 F8 ]
```

Since there are 5DF4h paragraphs of free memory, we now have no trouble allocating memory in TSRFILE.

"Porting" MEM.C to use the generic TSR was straightforward. We had to get rid of a call to calloc(), replace any calls to exit() with simple returns, replace any "\n" with "\r\n," and make a few other minor adjustments. The key change, however, was that we had to link with a version of printf() that doesn't call malloc(), because we blew away our near heap during TSR initialization.

This non-malloc version of printf() is provided in the module PUT.C, which we can be used with any program that uses the generic TSR. The non-malloc version of printf() uses the stdarg facilities of ANSI C, in particular the function vsprintf(), which can be used to easily create functions that take variable-argument lists. PUT.C also contains a number of other helpful functions. Prototypes for the functions appear (naturally) in PUT.H:

```
/* PUT.H -- STDERR output routines, no malloc */

// calls _dos_write, returns number of bytes actually writen
unsigned doswrite(int handle, char far *s, unsigned len);

// displays ASCIIZ string on STDERR
unsigned put_str(char far *s);

// displays character on STDERR
unsigned put_chr(int c);

// displays number (width, radix) on STDERR
unsigned put_num(unsigned long u, unsigned wid, unsigned radix);

// PUT includes alternate version of printf: goes to STDERR,
// doesn't use malloc. Same prototype as <stdio.h>

// get string from STDERR, returns actual length
unsigned get_str(char far *s, unsigned len);

#define putstr(s)       { put_str(s); put_str("\r\n"); }
#define put_hex(u)      put_num(u, 4, 16)
#define put_long(ul)    put_num(ul, 9, 10)


/* PUT.C -- STDERR output routines, no malloc */

#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include <stdarg.h>
#include <dos.h>
#include <bios.h>

#define STDERR          2

#include "put.h"

// returns length of far string
#ifdef _MSC_VER
#define fstrlen(s)      _fstrlen(s)     // MSC 6.0
#else
size_t fstrlen(const char far *s)       // MSC 5.1
{
    size_t len = 0;
    while (*s++) len++;
    return len;
}
#endif

unsigned doswrite(int handle, char far *s, unsigned len)
{
    unsigned bytes;
    _dos_write(handle, s, len, &bytes);
    return bytes;
}

unsigned put_str(char far *s)
{
    return doswrite(STDERR, s, fstrlen(s));
}

unsigned put_chr(int c)
{
    return doswrite(STDERR, (void far *) &c, 1);
}

#define putstr(s)   { put_str(s); put_str("\r\n"); }

#define MAX_WID     12

unsigned put_num(unsigned long u, unsigned wid, unsigned radix)
{
    char buf[MAX_WID+1], *p;
    int i, digit;
    if (wid > MAX_WID)
```

```
        return;
    for (i=wid-1, p=&buf[wid-1]; i >= 0; i--, p--, u /= radix)
    {
        digit = u % radix;
        *p = digit + ((digit < 10) ? '0' : 'A' - 10);
    }
    buf[wid] = 0;
    return doswrite(STDERR, (void far *) buf, wid);
}

#define put_hex(u)      put_num(u, 4, 16)
#define put_long(ul)    put_num(ul, 9, 10)

int _FAR_ _cdecl printf(const char _FAR_ *fmt, ...)
{
    static char buf[128];
    int len;
    va_list marker;
    va_start(marker, fmt);
    len = vsprintf(buf, fmt, marker);
    va_end(marker);
    return doswrite(STDERR, (void far *) buf, len);
}

unsigned get_str(char far *s, unsigned len)
{
    extern void (interrupt far *old_int28)(void);
    unsigned rcount;

    /* give TSRs a chance by calling INT 28h */
    while (! _bios_keybrd(_KEYBRD_READY))
        (*old_int28)();

    if ((_dos_read(STDERR, s, len, &rcount) != 0) || (rcount < 3))
        return 0;
    s[rcount-2] = '\0';
    return rcount-2;
}
```

In MEM.C, the C preprocessor #ifdef statement was used to conditionally compile either a stand-alone or a TSR version. For example:

```
#ifdef TSR
void fail(char *s) { printf("%s\r\n", s); return; }
#else
void fail(char *s) { puts(s); exit(1); }
#endif
```

The changes needed to make MEM a pop up were all of a similar nature, and are so straightforward and uninteresting that we leave them as an exercise for the reader. In any case, the resulting TSRMEM.EXE can be found on the disk that accompanies this book.

TSR2E

Finally, we jump the gun a little bit by porting a program from the next chapter in this book. As Jim Kyle explains there, INT 2Eh is the "backdoor" to the DOS command interpreter. The TEST2E command interpreter from chapter 6 can be easily turned into a TSR: an instant pop-up copy of COMMAND.COM!

This really does work. The only peculiarity is that on occasion when we pop up TSR2E and type in a command, we get the following message from the resident portion of COMMAND.COM:

```
Memory allocation error
Cannot start COMMAND, exiting
```

Note, however, that this is different from the horrifying message one sees when the MCB chain has been trashed:

```
Memory allocation error
Cannot load COMMAND, system halted
```

The message "exiting" rather than "system halted" is for real. If we just try to execute the command again, it works. In any case, this should probably join the list of other INT 2Eh caveats found in the next chapter.

We can use TSR2E, not only to issue internal commands (such as DIR or COPY), but also to issue external commands that launch other programs or even batch files.

As shown in the makefile presented earlier in the chapter, we build TSR2E by combining the generic TSR components with Jim Kyle's three files, SEND2E.C, HAVE2.ASM, and DO2E.ASM. These files are unchanged. All changes for going TSR are confined to the module TEST2E.C, where we change the name of the module entry point from main() to application(), add a test that lets TSR2E avoid popping up when COMMAND.COM is already running, and use the facilities in the PUT module rather than the C standard library's mallocy stdio facilities. Here is the altered version of TEST2E.C:

```
/* TEST2E.C -- version to build TSR2E */

#include <stdlib.h>
#include <string.h>
#include <dos.h>

#include "put.h"

#define MK_FP(seg,ofs) \
    ((void far *)(((unsigned long)(seg) << 16) | (ofs)))

extern unsigned foreground_psp;     // in TSREXAMP.C
extern int Send2E(char *command);   // in SEND2E.C

static char buf[80];
static int running = 0;

typedef enum { SAVE=0, RESTORE } SAVEREST;
typedef void (interrupt far *INTVECT)();

void interrupts(int restore)
{
    static INTVECT int_1b, int_23, int_24;
    if (restore)
    {
        _dos_setvect(0x1b, int_1b);
        _dos_setvect(0x23, int_23);
        _dos_setvect(0x24, int_24);
    }
    else
    {
        int_1b = _dos_getvect(0x1b);
        int_23 = _dos_getvect(0x23);
        int_24 = _dos_getvect(0x24);
    }
}

void application(void)
{
    // don't run if we are already running
    if (running)
        return;
    running++;

    // don't execute INT 2Eh if COMMAND.COM already running
    // see if COMMAND.COM running by checking if current PSP is the
    // same as its own parent
```

```
    if (foreground_psp ==
        *((unsigned far *) MK_FP(foreground_psp, 0x16)))
    {
        put_str("COMMAND.COM already running");
        running--;
        return;
    }

    put_str("TSR COMMAND SHELL: type DOS commands, or BYE to quit\r\n");
    for (;;)
    {
        put_str("$ ");
        if (! get_str(buf, 80))
            break;
        if (strcmp(buf, "bye") == 0 || strcmp(buf, "BYE") == 0)
            break;
        interrupts(SAVE);
        Send2E(buf);
        interrupts(RESTORE);
    }
    putstr("Bye");
    running--;
}
```

Note that we save and restore the Ctrl-C, Ctrl-Break, and Critical Error interrupts around the call to Send2E(). With this precaution, even Ctrl-C, Ctrl-Break, and Critical Errors are handled properly within the INT 2Eh pop-up:

```
TSR COMMAND SHELL: type DOS commands, or BYE to quit
$ dir a:

Not ready error reading drive A
Abort, Retry, Fail? a

$ dir *.c /w

 Volume in drive C is RAMANUJAN
 Directory of  C:\UNDOC\RMICHELS

DOSSWAP  C      EXTERR   C      ^C

$ bye
Bye
```

One note of caution, however: don't try to install a TSR from within the pop-up command interpreter: it will hang your system sometime after you exit the pop-up.

Multitasking TSR

Finally, let us discuss TSRs that *don't* pop up at a user hotkey, but which do their work in the background. We call such programs multitasking TSRs to distinguish them from pop-ups. MULTI.C is a multitasking TSR shell that can be modified to perform a multiple of background tasks, from disk file copying to background communications.

The example presented here is an enhancement to the DOS PRINT utility. It periodically (activated by INT 8 and INT 28h) searches a \SPOOL directory for files having the extension .SPL. When a match is found, the TSR uses PRINT's INT 2Fh Function 01h interface, to ask PRINT to print the file. Once the file has been submitted for printing, the TSR periodically obtains a status report from PRINT. If the file is no longer in the print queue (its printing is complete), the file is deleted:

```
C:\UNDOC> print
C:\UNDOC> multi
C:\UNDOC> copy \undoc\rmichels\*.asm \spool\*.spl
C:\UNDOC> dir \spool

 Volume in drive C is RAMANUJAN
 Directory of  C:\SPOOL

.            <DIR>      3-23-89   9:54p
..           <DIR>      3-23-89   9:54p
TSRUTIL  SPL     5852   9-17-90  11:40p
STACK    SPL     1758   9-17-90  11:36p
        4 File(s)     94208 bytes free

C:\UNDOC> print

     C:\SPOOL\TSRUTIL.SPL is currently being printed

C:\UNDOC> dir \spool

 Volume in drive C is RAMANUJAN
 Directory of  C:\SPOOL
```

```
.            <DIR>      3-23-89   9:54p
..           <DIR>      3-23-89   9:54p
STACK    SPL      1758   9-17-90  11:36p
        3 File(s)    102400 bytes free
```

Basically, the program manages two independent processes, the foreground process and the background process. When it is time for one process to start, the current process is suspended, its registers saved on the stack for later restart. Once the TSR has been loaded for later restart, the environment of the suspended process' environment is restored, and it continues where it left off.

This multitasking is achieved by maintaining a count based on the timer interrupt. Each process get a specific amount of time; in the example above, the foreground process gets the most time so as not to degrade performance. It is possible to add more tasks, but you would need to maintain a list of SS:SP sets in order to service all running processes.

Most of the code is similar to TSREXAMP. The two main differences are that activation is via the timer interrupt, not a hot key, and that instead of completing its work during activation, the TSR is suspended for later restart. Because DOS is not reentrant, the TSR still must follow the rule of not interrupting DOS when it is active.

In most computer systems, multitasking is a method of quickly switching from one task to another, such that the computer appears to be running multiple tasks at the same time. Of course, it's not as simple as that. Multitasking systems are designed so that resources (such as disks, screens, and keyboards) can be shared by multiple applications. True multitasking systems such as OS/2 supply interface routines that are reentrant. The code segment of each routine has only one instance, and this code segment can be shared by multiple processes at the same time. Each user (each routine) will have its own instance of data. But, as we know, the INT 21h API is not like this, so our multitasking example is controlled to some extent by the DOS flags.

Task Switching

Every process has what can be called its context or frame. This consists of the following items:

- Register values (including code, data, and stack segment values)
- Program Segment Prefix (PSP), or process ID
- Disk Transfer Address
- Extended Error Information

During a task or context switch, these items must be saved and replaced by ones that pertain to the new task. In our example, we are simply flip-flopping between two tasks. If more independent tasks were required, we would need to keep a list of information for each task. Each item in the list might contain the relevant DOS information (PSP, DTA, etc.) and a pointer to the process stack segment and offset. The code to perform the list management is more complex and is not presented here. Also recall that we could use the DOS SDA.

The type of multitasking presented here is called *time-slicing*, because each task gets a predetermined slice of time in which to run. If needed, more intelligence could be added that would control the percentage of time each process gets. This could be based on usage of the operating system (INT 21h) or disk (INT 13h). We could chain these interrupts, and if a process is making extensive use of these resources, we could lower its time slice to give other processes more time.

This method is used during our background process. If the file being printed is still in the print queue, a timer limit variable is set so that background processing will terminate immediately. There is no point in running in the background if the program is simply waiting for the print spooler to complete its job.

MULTI Installation

MULTI installs using the techniques already described. Some interrupt-handling routines must be in assembly language, because they augment the normal interrupt process. Upon return from most interrupts, the flags are restored to their condition just before the interrupt was invoked. But three of the interrupts we are interested in handling—INT 13h, 25h, and 26h—do not following this convention. INT 13h (like INT 21h) returns error conditions via flags, so an INT 13h handler must end with a RET 2 rather than an IRET. Meanwhile, the DOS absolute disk routines, INT 25h and 26h, leave the flags on the stack, and so exit with a RET rather than an IRET. This code is marked as IFDEF MULTI in TSRUTIL.ASM, shown earlier in the chapter.

Another difference between TSREXAMP and MULTI is that, in MULTI, the address of our main_loop function is placed on the stack that is created in the main() procedure. This is typical of multitasking code, and causes execution to begin at this address when the background process is first activated.

Timer Interrupt

MULTI is activated, not by a user keypress of course, but by the INT 8 timer tick interrupt. MULTI's INT 8 handler increments a variable called tic_count, which keeps track of how many timer ticks have occurred. Each process that we manage is allowed a given number of ticks. Once the current tick count exceeds the process limit, and if the INDOS flag is zero (except within INT 28h, in which it will be one, but must not be more than one) and the unsafe_flag explained below is FALSE, we suspend that process and activate the other process.

At this point, we can suspend the process. Depending upon what process we are currently executing, we call either suspend_background or suspend_foreground. If we are suspending the foreground, we set the stack to be our local TSR stack. In either case, we also carry out the save/restore regime used earlier in tsr_function in TSREXAMP: setting INT 1Bh, 23h, and 24h, and swapping PSP, DTA, and extended error information.

Upon return to the new_int8 function, it restores the stack, if we have just activated the foreground process. It continues along the INT 8 interrupt chain.

One special condition applies the first time we activate our background process. We have never interrupted this process, so its registers are not on the stack. Because of this, an assembly-language function, timer_int_chain(), is called to jump to the next timer interrupt handler.

We earlier referred to a flag variable called unsafe_flag. Since there is no INBIOS flag we can use, we have to create our own, and unsafe_flag is set TRUE when critical BIOS services (INT 10h video and INT 13h disk), or the DOS absolute disk services (INT 25h and 26h) are in progress.

Idle Interrupt

The timer interrupt is not the only way we can run in the background. We also use the DOS idle interrupt (INT 28h), discussed earlier. This will allow us to continue processing while the system is sitting at the COMMAND prompt. Otherwise, we would never run in the background while COMMAND was awaiting orders.

During an INT 28h, if the Int28DosBusy function returns FALSE and the background is not already active, we set the foreground limit to zero and the int_28_active variable to TRUE. We then wait for the int_28_active variable to go false before continuing. This allows a nearly immediate task switch to the background process.

Keyboard Interrupt

Notice that we have installed a service routine for INT 9, the keyboard hardware interrupt. Whenever the user presses a key, the background task time limit is set to 0 (causing the task to go into a suspended state more quickly). This gives the user better response time.

Printing

The main_loop() function in our example is the background process. This function performs the work of searching for files and submitting them to the spooler via INT 2Fh Function 01h.

Notice that the example includes numerous loops that do nothing during the main_loop() function. This is to avoid constant disk access by the background task. The background time limit is also set to zero in a number of places, to ensure that the background becomes suspended at that time.

MULTI.C

The following is MULTI.C, source code for the multitasking TSR. To build MULTI.EXE, use the instructions found in the makefile shown earlier in this chapter.

```
/* MULTI.C */

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include <io.h>
#include "tsr.h"

#define SEARCH_DIR   "C:\\SPOOL\\"
#define STACK_SIZE  4096    /* must be 16 byte boundary */
#define SET_DTA     0x1a  /* SET Disk Transfer Address */
#define GET_DTA     0x2f  /* GET Disk  Transfer Address */

#define BACKGROUND_TICS 2
#define FOREGROUND_TICS 16
#define BACKGROUND_YIELD 0
#define FOREGROUND_YIELD 0
```

```
struct  prReq
{
    char level;
    char far *fname;
};

char    far *stack_ptr;   /* stack for our background TSR */
char    far *ptr;
unsigned ss_save;         /* slot for stack segment register */
unsigned sp_save;         /* slot for stack pointer register */
unsigned unsafe_flag = 0; /* set true by various interrupts */
int first_time = 1;       /* flag for first time in running background
*/

int my_psp;               /* our TSR's psp */
int foreground_psp;       /* PSP of interrupted foreground process */
int foreground_dta_seg;   /* DTA of interrupted foreground process */
int foreground_dta_off;
int ctr=0;
int tic_count = 0;        /* counts timer tices */
int in_progress = 0;      /* true if we're in background process */

char search_work[65];
struct ExtErr my_ErrInfo;
struct ExtErr foreground_ErrInfo;

int foreground_limit = FOREGROUND_TICS; /* foreground cycle limit */
int background_limit = BACKGROUND_TICS; /* background cycle limit */

char search_dir[65] = {SEARCH_DIR}; /* dir to search for spool files */
volatile int   int_28_active = 0;   /* true if activated by INT 28 */
volatile int   interval_timer;      /* for sleeping a number of tics */

/* old interrupt pointers are stored here */
INTVECT old_int8, old_int9, old_int10, old_int13;
INTVECT old_int1B, old_int23, old_int24, old_int25;
INTVECT old_int26, old_int28;

/* prototypes for this module */
void main_loop();
void interrupt far new_int8(INTERRUPT_REGS);
void interrupt far new_int9(INTERRUPT_REGS);
void interrupt far new_int10(void);
void interrupt far new_int13(void);
void interrupt far new_int1B(INTERRUPT_REGS);
void interrupt far new_int23(INTERRUPT_REGS);
void interrupt far new_int24(INTERRUPT_REGS);
```

```
void interrupt far new_int25(void);
void interrupt far new_int26(void);
void interrupt far new_int28(INTERRUPT_REGS);
int  spooler_active(void);
int  search_spl_que(char * fname);
void suspend_foreground(void);
void suspend_background(void);

/* returns nonzero if PRINT installed */
int spooler_active()
{
    union REGS  regs;

    regs.x.ax = 0x0100;          /* PRINT install check */
    int86(0x2f,&regs,&regs);     /* call multiplex interrupt */
    return(regs.h.al == 0xff);   /* FF if installed */
}

/* returns nonzero if file is in the spooler queue */
int search_spl_que(char * fname)
{
    union REGS  regs;
    struct SREGS   sregs;
    char far *  que_ptr;
    char que_name[65];
    int i;
    int found = 0;

   if (spooler_active())
   {
      regs.x.ax = 0x0104;  /* get spooler status */
      int86x(0x2f,&regs,&regs,&sregs);
      /* on return from call DS:SI points to print queue */
      FP_SEG(que_ptr) = sregs.ds;
      FP_OFF(que_ptr) = regs.x.si;
      /* release hold on spooler, side effect of status*/
      regs.x.ax = 0x0105;
      int86x(0x2f,&regs,&regs,&sregs);
      while (*que_ptr && !found)  /* while items in queue */
      {
         for (i = 0; i < 65; i++)
            que_name[i] = *(que_ptr + i);
         if (found = !strcmpi(que_name,fname))
             break;
         que_ptr += 65;
      }
   }
```

```
   return(found);
}

void main_loop()
{
    struct find_t c_file;
    union REGS regs;
    struct SREGS sregs;
    struct prReq prRequest;
    struct prReq far * ptr;
    int sleep_cntr;

    while (1)
    {
      strcpy(search_work,search_dir);
      strcat(search_work,"*.SPL");   /* create dir search string */

      interval_timer = 18 * 30;     /* search every 30 seconds */
      while (interval_timer)    /* wait between each dir search */
          background_limit = BACKGROUND_YIELD;  /* yield for fgrnd */

      if (!_dos_findfirst(search_work,_A_NORMAL,&c_file))
      {
         /* if spooler installed, dos 3.xx+ and file size > 0 */
         if (spooler_active() && _osmajor >= 3 && c_file.size)
         {
           strcpy(search_work,search_dir);
           strcat(search_work,c_file.name); /* full pathname */
           prRequest.level = 0;
           prRequest.fname = search_work;
           regs.x.ax = 0x0101;
           ptr = &prRequest;
           sregs.ds = FP_SEG(ptr);
           regs.x.dx= FP_OFF(ptr);
           int86x(0x2f,&regs,&regs,&sregs);

           while (search_spl_que(search_work))  /* wait till done */
           {
               interval_timer = 18 * 30;  /* sleep for 30 seconds */
               while (interval_timer)
                   background_limit = BACKGROUND_YIELD;
           }

           unlink(search_work);     /* delete file */
           background_limit = BACKGROUND_YIELD;
         }
      }
```

```
    }
}

union REGS regs;
struct SREGS sregs;

void suspend_foreground()
{
    /* SWAP TO BACKGROUND */
    tic_count = 0;
    /* save old handlers */
    old_int1B= _dos_getvect(0x1B);
    old_int23= _dos_getvect(0x23);
    old_int24= _dos_getvect(0x24);

    /* set our interrupt handlers */
    _dos_setvect(0x1b,new_int1B);
    _dos_setvect(0x23,new_int23);
    _dos_setvect(0x24,new_int24);

    /* save current PSP and set to ours */
    foreground_psp = GetPSP();
    SetPSP(my_psp);

    /* get foreground DTA */
    regs.h.ah = GET_DTA;
    intdosx(&regs, &regs, &sregs);
    foreground_dta_seg = sregs.es;
    foreground_dta_off = regs.x.bx;

    /* set up our DTA */
    regs.h.ah = SET_DTA;
    regs.x.dx = 0x80;   /* use default in PSP area */
    sregs.ds = my_psp;
    intdosx(&regs, &regs, &sregs);

    /* save error info */
    GetExtErr(&foreground_ErrInfo);

    if (! first_time)
        SetExtErr(&my_ErrInfo);

    in_progress = 1;
    background_limit = BACKGROUND_TICS; /* set default limit */
}

void suspend_background()
```

```
{
    /* SWAP TO FOREGROUND */

    /* put back original DTA */
    regs.h.ah = SET_DTA;
    regs.x.dx = foreground_dta_off;
    sregs.ds  = foreground_dta_seg;
    intdosx(&regs, &regs, &sregs);

    /* put back original PSP */
    SetPSP(foreground_psp);

    /* put back original INTS */
    _dos_setvect(0x1b,old_int1B);
    _dos_setvect(0x23,old_int23);
    _dos_setvect(0x24,old_int24);

    /* get error info */
    GetExtErr(&my_ErrInfo);
    SetExtErr(&foreground_ErrInfo);

    tic_count = 0;
    in_progress = 0;
    int_28_active = 0;
    foreground_limit = FOREGROUND_TICS; /* set default limit */
}

/**********
* TIMER TICK INTERRUPT HANDLER
**********/
void interrupt far new_int8(INTERRUPT_REGS r)
{
    tic_count++;

    if (interval_timer)
        interval_timer--;

    if ((in_progress && (tic_count >= background_limit) &&
       !DosBusy() && !unsafe_flag) ||
       (in_progress && int_28_active && !Int28DosBusy() &&
           (tic_count >=background_limit)))
    {
      suspend_background();
      restore_stack();
    }
    else if ((!in_progress && (tic_count >= foreground_limit) &&
        !DosBusy() && !unsafe_flag) ||
```

```
        (!in_progress && int_28_active && !Int28DosBusy() &&
            (tic_count >=foreground_limit)))
    {
      set_stack();
      suspend_foreground();
      if (first_time)
      {
         first_time = 0;
         timer_int_chain();
      }
    }
   old_int8();    /* call old handler */
}

/**********
* KEYBOARD INTERRUPT HANDLER
**********/
void interrupt far new_int9(INTERRUPT_REGS r)
{
   unsafe_flag++;
   old_int9();
   if (in_progress)
      background_limit = BACKGROUND_YIELD; /* set to swap to fgrnd */
   foreground_limit = 18;               /* since user hit keyboard */
   unsafe_flag--;
}

/*********
* CTRL-BREAK INTERRUPT HANDLER
*********/
void interrupt far new_int1B(INTERRUPT_REGS r)
{
    /* do nothing */
}

/**********
* CTRL-C INTERRUPT HANDLER
**********/
void interrupt far new_int23(INTERRUPT_REGS r)
{
    /* do nothing */
}

/**********
* CRTITICAL ERROR INTERRUPT HANDLER
**********/
void interrupt far new_int24(INTERRUPT_REGS r)
```

```
{
    if (_osmajor >= 3)
        r.ax = 3;   /* fail dos function */
    else
        r.ax = 0;
}

/**********
* DOS IDLE INTERRUPT HANDLER
**********/
void interrupt far new_int28(INTERRUPT_REGS r)
{
    if (!in_progress && !Int28DosBusy() && !unsafe_flag &&
        tic_count > foreground_limit)
    {
      foreground_limit = FOREGROUND_YIELD;   /* stop foreground */
      int_28_active = 1;
      _enable();              /* STI */
      while (int_28_active)
          ;        /*spin waiting for task swap to bckgrnd*/
    }
   (*old_int28)();    /* call old handler */
}

main()
{
    unsigned        memtop;
    unsigned        dummy;
    void far* far*  tmpptr;

    puts("Multi-Tasking PRINT spooler installing");

    if (_osmajor < 3)
    {
        puts("Error: MS-DOS version 3.00 or greater required");
        exit(1);
    }

    if (! spooler_active())
        puts("Warning: Print Spooler not active");

    InitInDos();
    my_psp = GetPSP();

    /* MALLOC a stack for our TSR section */
    stack_ptr = malloc(STACK_SIZE);
    stack_ptr += STACK_SIZE;
```

```
    ptr = stack_ptr;
    *(--stack_ptr) = 0xF2;  /* set up stack as if an an IRET was done*/
    *(--stack_ptr) = 0x02;
    stack_ptr -= 4;
    tmpptr = stack_ptr;
    *(tmpptr) = main_loop;

    /* get interrupt vectors */
    old_int8  = _dos_getvect(0x08); /* timer int */
    old_int9  = _dos_getvect(0x09); /* keyboard int */
    old_int10 = _dos_getvect(0x10); /* video int */
    old_int13 = _dos_getvect(0x13); /* disk int */
    old_int25 = _dos_getvect(0x25); /* sector read int */
    old_int26 = _dos_getvect(0x26); /* sector write int */
    old_int28 = _dos_getvect(0x28); /* dos idle int */

    init_intr();    /* init asm variables */

    _dos_setvect(0x08,new_int8);
    _dos_setvect(0x09,new_int9);
    _dos_setvect(0x10,new_int10);
    _dos_setvect(0x13,new_int13);
    _dos_setvect(0x25,new_int25);
    _dos_setvect(0x26,new_int26);
    _dos_setvect(0x28,new_int28);

#define PARAGRAPHS(x)   ((FP_OFF(x) + 15) >> 4)

    /* release unused heap to MS-DOS */
    /* All MALLOCS for TSR section must be done in TSR_INIT() */
    /* calculate top of memory, shrink block, and go TSR */
    segread(&sregs);
    memtop = sregs.ds + PARAGRAPHS(ptr) - _psp;

    _dos_setblock(memtop, _psp, &dummy);
    _dos_keep(0, memtop);
}
```

When compiling the MULTI TSR example under Microsoft C 6.0 we ran across a code-optimization "gotcha." Notice that the new_int28() function simply sets up the tic counting variables so that the background will become active. The code then sets the int_28_active semaphore and waits for it to be cleared:

```
int_28_active = 1;
while (int_28_active)
    ;
```

The idea behind this code was to wait until a task swap occurred. During the task swap, int_28_active is set to zero.

But when compiling with full optimization, the compiler sees that that variable is 1, and is not altered in the while loop. Therefore it figures this is an infinite loop, and generates a JMP $. What it does not know is that the timer interrupt routine will clear this flag.

To avoid this problem, but still allow optimization, we declare the int_28_active variable with a "volatile" attribute:

```
volatile int int_28_active;
```

This ANSI C keyword tells the compiler that the variable may change from an external source.

This multitasking TSR is a simple example to which a few enhancements could be added. Memory could be swapped to disk or EMS as needed. If keyboard and CRT I/O is required by the background task, you must be careful to save and restore the appropriate settings. You also must not switch tasks while in the middle of a BIOS Video service. For a true multitasking system, MS-DOS alone is probably not the way to go. Commercial products (windowing systems such as Windows or DesqView, MS-DOS replacements such as PC-MOS or VM/386, or OS/2) will give you much more functionality. But such systems use many of the same principles outlined here.

Chapter 6

Command Interpreters

Jim Kyle

Every operating system that permits more than a single program to run requires some sort of command interpreter. In the earliest days, commands were interpreted by the human operator, who picked out the correct plugboard or card deck and set the system into action. Now, the job is done by a program (often called the *shell* because it surrounds the system kernel, but more formally termed the *command interpreter*) that prompts the user for input and then reacts to that input.

For most users, a command interpreter is the closest contact they ever have with an actual operating system. The familiar "C>" prompt from COMMAND.COM, the character-based, command-line shell that comes with MS-DOS, is almost universally called "the DOS prompt" although the interpreter is not in fact an integral part of MS-DOS itself. Alternate interpreters are available.

This chapter examines first the functional requirements that must be met by any command interpreter. While examining these requirements, the chapter explores several undocumented services that DOS provides to simplify the task of interfacing with the command interpreter. The first section concludes with a tiny shell program you can use to replace COMMAND.COM; this program illustrates exactly what the requirements are for creating a command interpreter.

With the functional requirements established, the chapter continues by dissecting COMMAND.COM to see how it meets those requirements. In this section, you'll learn about the environment, how it works, and how to locate COMMAND.COM once it is loaded into memory. This section includes a number of utility routines for locating and dealing with environment blocks.

Next, we examine some alternative command interpreters that are now available, together with some "shells" that are actually only extensions to COMMAND.COM.

The chapter concludes with a sample program that combines the use of documented and undocumented features to permit editing of the master environment. This program, ENVEDT, works with any command interpreter that supports the undocumented DOS hook INT 2Eh; actually, the command interpreter need not provide full support, so long as it includes a minimal interrupt handler for the service.

Several notorious undocumented aspects of the DOS programmer's interface are covered in this chapter, including the "backdoor" to the command interpreter (INT 2Eh) and the DOS master environment block. The less well-known but quite important installable-command interface (INT 2Fh Function AEh) is also discussed in detail. The chapter contains large amounts of sample code to illustrate all these topics.

Experienced DOS "power users" who are not programmers, and who might have been bewildered by other parts of this book, might find several areas of interest in the first part of this chapter. On the other hand, experienced DOS programmers might want to skip ahead to the section called "The Hooks MS-DOS Provides."

Requirements of a Command Interpreter

The major requirements for any command interpreter are as follows:

- To provide a means of obtaining commands from the human operator
- To interpret those commands
- To act upon them by dispatching appropriate processes

A fourth, implicit, requirement is that these actions be enclosed in a loop, so that more commands may be issued after all current commands have been processed.

Obtaining Operator Input

The most essential requirement for any command interpreter is that it have a means of obtaining commands to be interpreted, for without that nothing else has any meaning.

Operator input can be obtained from keystrokes entered directly by the user in response to a prompt from the command interpreter. There are, however, several alternate methods, all of them frequently used.

The DOS Prompt The command interpreter usually signals the operator that it's waiting for input by issuing a prompt message, usually known simply as "the prompt." Some menu-style shell programs turn the cursor or the mouse pointer on to indicate that input is needed, but more often these programs display their menus only when seeking input, so that the mere presence of the menu on the screen serves as the prompt.

In the more conventional command-line operation, though, the prompt consists of a relatively short sequence of characters. The default prompt message of COMMAND.COM is simply "C>", where "C" indicates the drive letter of the current drive and ">" is simply a visual delimiter.

Virtually all command interpreters, though, give the user a means to modify the prompt into whatever might be desired. Although a dedicated PROMPT command is used to define custom prompt messages, the actual message is stored as a string in the environment space and can be changed in the same manner as any other environment string. Many hard disk users use PROMPT pg rather than the default C> prompt (equivalent to PROMPT ng). In addition, ANSI.SYS can be used to put the current drive and directory at the top left corner of the screen, for example, while the "usual" prompt appears in its normal place.

Keystrokes The "normal" source of input to the command interpreter is the system keyboard, but it's examined only as a last resort, if the interpreter is unable to get input from any of the alternates!

This apparently backward approach actually has a most logical basis; it lets you start a job using the alternate methods and then switch to the keyboard to finish the job. This method can, however, be highly confusing to the neophyte user. Let's defer examination of the possible confusion until we've seen what alternates to the keyboard exist.

When the keyboard furnishes input, most command interpreters use the standard DOS Read String function (Int 21h, Function 0Ah) to obtain that input.

This function provides a rudimentary string-edit capability, and INT 21h Function 0Ah furnishes additional editing via several of the function keys. Unlike older operating systems, keyboard input to the command interpreter in DOS is not forced to uppercase but is passed to the interpreter exactly as you typed it, except for characters erased via the backspace keys. Command-line editors, such as Chris Dunford's CED, hook INT 21h and supply their own Read String Function, thereby enhancing the editing of *any* program that calls Function 0Ah, not just COMMAND.COM.

All command interpreters used with MS-DOS have a size limit of 126 characters for their keyboard input. This limit is imposed by the layout of the PSP (refer to chapter 3), which allows only 128 bytes for the command tail. Of these 128 bytes, one is taken by the character count and one by the CR character that terminates the input string.

Although it would be possible to extend this limit by a few bytes, because the command itself is never copied into the PSP, no actual interpreter does so. For simplicity, they all provide a maximum 128-byte input buffer.

Batch Files In many applications, a relatively complex series of commands must be entered to get the desired action started. Batch files provide the most generally used method for supplying those commands. You type the commands into the batch file once, and then the entire sequence is pumped into the command interpreter when you invoke the batch file.

Command interpreters treat a batch file as a special type of external command. Although different interpreters process these files in different ways, the general idea is that the interpreter reads the file one line at a time, then executes the command contained on that line before coming back to read the next.

For safety, COMMAND.COM closes the batch file each time a line is read, and reopens it to read the next line. This makes it possible to use batch files with a single-drive system, by having a copy of the file on each diskette; so long as all copies are named the same and have the same content, the command interpreter will never be aware that the disks were swapped between lines.

A special batch file is executed by COMMAND.COM (and by all compatible command interpreters) at system start-up; this file, always named AUTOEXEC.BAT, normally contains the commands necessary to customize your system and install any TSRs you use. It is invoked any time that COMMAND.COM is executed with its "/P" (permanent) option switch; normally this happens only as part of the system boot-up sequence.

In addition to AUTOEXEC.BAT, another frequent use for batch files is to create menuing systems. The simplest form of such systems ECHOes the menu choices to the screen and requests input of the corresponding number. For each number, a batch file such as "1.BAT" is created to carry out the necessary command sequence.

From that basic starting point, you can go as far as you like. One of the main features of MS-DOS 4.x, the DOSSHELL capability, is essentially a batch-file-based menu system, though it uses special undocumented hooks in DOS itself to simplify its actions. We'll look at it in more detail later in this chapter.

Still another variant of the menuing system is the dispatching program, which operates much like a menuing system, except that, rather than using a multiplicity of batch files, it stores the necessary data in a special data file and then dispatches the chosen process by means of the DOS EXEC function.

Batch Enhancers and Compilers Batch files have become so popular that several firms offer "batch language enhancement" programs, such as BE (Batch Enhancer) in the Norton Utilities, EBL (Extended Batch Language), and several *PC Magazine* utilities such as Michael Mefford's BATCHMAN. These programs are widely used, and many users find them extremely valuable.

Recently, batch file compilers have become popular as well. In addition to Wenham Software's BATCOM and Hyperkinetix's The Builder, *PC Magazine* (August 1990) has published a batch file compiler, Doug Boling's BAT2EXEC. These compilers turn .BAT files into true .COM files.

This raises an interesting issue: as is well known, the SET statement in a .BAT file will alter the "master environment" (explained later in the section "How COMMAND.COM Uses the Environment"), but a seemingly equivalent attempt to change the environment from a .COM or .EXE program results only in an alteration to the program's *local copy* of the environment, which gets thrown away when the program exits.

How then can the proper semantics of the SET statement be preserved when a .BAT file is compiled into a .COM file? Simple: the compiled SET statement uses undocumented DOS to alter the master environment. For example, any time a .BAT file with a SET statement is compiled with BAT2EXEC, the resulting .COM file calls undocumented INT 21h Function 52h. Why Function 52h? To get a pointer to the DOS List Of Lists which contains (at offset -2) the segment of the first MCB (see Chapter 3). By walking the MCB chain, the program can find the master environment. This is explained in much greater detail (including a prob-

lem with BAT2EXEC) later on, in the section "Other Ways of Locating the Environment." In any case, the growing popularity of batch file compilers will, for better or worse, probably produce a proliferation of programs that rely on undocumented DOS functions and data structures.

"Losing" Stuffed Commands All batch file processing is performed by the command interpreter, not by DOS itself. This means that only the command interpreter can obtain input directly from the batch file; it's not possible to provide input directly to your programs by lines typed into any batch file. Although a batch file is meaningful only to the command interpreter, however, the keyboard buffer applies to both the command interpreter and to any commands (whether internal like COPY, or external like 123) that may be invoked by the interpreter's action. This means that programs can "stuff" your desired input into the keyboard buffer, from which other programs can retrieve it. This is useful with programs such as 1-2-3 that (still!) do not take command-line arguments.

One such keyboard stuffer is Charles Petzold's KEY-FAKE.COM, available in the book *PC Magazine DOS Power Tools*. KEY-FAKE is used here to illustrate a feature of batch files that, although seemingly obvious, appears to cause a lot of confusion. Any other keyboard stuffer will serve equally well.

The following batch file creates a file called FOO.BAR, containing the single line "hello" (13 26 13 emits a newline, ^Z, newline sequence):

```
echo off
key-fake "hello" 13 26 13
copy con foo.bar
```

Note that input is stuffed into the keyboard buffer *before* the COPY CON command is invoked. If you want to stuff both the input *and* the command into the keyboard, however, the command must go first:

```
echo off
key-fake "copy con foo.bar" 13 "hello" 13 26 13
```

So far so good. Now let's add a line to the end of the batch file:

```
echo off
key-fake "copy con foo.bar" 13 "hello" 13 26 13
echo Done creating FOO.BAR
```

What happens when we run this? The second command is issued *before* the first command:

```
C:\UNDOC\KYLE>tmp
Done creating FOO.BAR
C:\UNDOC\KYLE>copy con foo.bar
hello
^Z
    1 File(s) copied
```

The message that signals that the operation is complete is displayed before the operation begins! Just by adding a line to the end of the batch file, we somehow caused the COPY CON command to be deferred.

It gets worse. Do the keyboard stuffing inside a loop, and the COPY CON command *never* gets executed, resulting in an infinite loop:

```
echo off
del foo.bar
:loop
key-fake "copy con foo.bar" 13 "hello" 13 26 13
if not exist foo.bar goto loop
```

Simply putting a command inside a loop causes the batch file to stop working! What is going on around here?

If, as a final experiment, we return to our original idea of putting the COPY CON command itself in the batch file, and stuffing only its *input* into the keyboard buffer, everything starts working properly again:

```
echo off
del foo.bar
:loop
key-fake "hello" 13 26 13
copy con foo.bar
echo Done creating FOO.BAR
if not exist foo.bar goto loop
```

What we have just seen merely illustrates that the command interpreter exhausts all batch file lines before looking in the buffer for keyboard input. Therefore, unless the keyboard stuffer happens to execute as the last command in the batch file, the stuffed command isn't executed at the correct time.

This detail of command interpreter operation seems rather obvious, yet it spawns at least one question per week on the major information network forums that deal with hardware and software problems. The rule to follow in order to avoid the problem is simple: invoke commands directly from the batch file, not by stuffing the keyboard buffer; provide only program input via the buffer.

Interpreting Operator Requests

Once the operator's input has been obtained, it must be put into a form acceptable to MS-DOS (that is, it must be parsed for file names, etc.), interpreted, and acted upon. This section first describes the factors involved in parsing the input, then discusses how it is interpreted, and finally deals with the execution of internal commands. Any input not recognized as an internal command is passed to the dispatching procedure as a possible "external" command that must be loaded from a file to be executed.

Parsing for Inclusion in the PSP The "standardized" parsing done by DOS command interpreters traces directly back to CP/M; the major difference is MS-DOS includes INT 21h Function 29h, fully documented, to perform the parse for you.

In this standardized parse, certain characters are treated as "white space" separators. These include the blank space itself, the tab character, the "switch character" (normally a forward slash, but in some versions of DOS this can be changed to a hyphen: see below), the comma, the colon, the semicolon, and the equal sign. Several of these have additional syntactic significance, but all are recognized as marking the end of the possible command name.

The parse begins by skipping over all blank or tab characters at the front of the input line. When a nonblank character is found, it is converted to uppercase if it's alphabetic and moved to an internal parse buffer. From that point until one of the white space characters is encountered, all characters are moved to the parse buffer and case-converted if necessary. When the terminating white space character is found, the parse pointer is left pointing to it.

All remaining characters from the input buffer will be moved to the "command tail" area of the new process' PSP, starting at offset 81h and the count of those characters (omitting the terminating CR) is stored at offset 80h. None of these characters is case-converted during the move.

Next, the first complete word (if any) in the command tail is examined to determine if it can be a filename. That is, it must contain no characters that would not be valid in a filename; its second character can be a colon and any subse-

quent character up to the 9th can be a period. This permits the CP/M and DOS V1 (nondirectory) file specification, such as "A:FILENAME.EXT", to be accepted. If the word passes all these tests, it is converted to FCB format (which includes case conversion) and is then filled into the FCB1 region of the PSP, at offset 5Ch. The drive code corresponding to the drive letter, if any, goes into the first byte, followed by the filename portion (padded to 8 bytes with spaces if necessary). The period, like the colon, is omitted, and the extension goes into the next 3 bytes, again padded with spaces.

When this is complete, the process is repeated for the second complete word of the command tail, to fill in the FCB2 area of the PSP at offset 6Ch.

This is a direct copy of the steps performed by CP/M programs, and the syntax of many of the older internal commands is based on these parsing rules. For instance, RENAME originally accepted only filenames, and the PSP layout of FCB1 and FCB2 required only that one pointer be set up in the CPU registers before the command passed control to DOS to do the renaming.

In DOS 1.x, many programs took advantage of these parsing rules to extract their first two command-line arguments from the filename fields of FCB1 and FCB2. By doing so, they could avoid the need to furnish their own parsing routines; the command interpreter had already done the work for them.

However, these routines are not capable of handling subdirectory references and full path names, so with DOS 2.x their usefulness began to fade, and today they are primarily a footnote to history. Unfortunately, some programs still depend on them, showing the persistence to this day of CP/M vestiges.

Upon completion of this standard parse, then, the command interpreter will have in an internal parsing buffer the first word of input converted to uppercase, and it will have in the new current PSP the two FCB areas and the command tail data. The next step is to determine whether the input was actually a valid command.

SWITCHAR If you've ever needed to switch between DOS and Unix machines, you may have been annoyed that whereas UNIX uses the forward slash (/) for paths and hyphens (-) for command-line options, MS-DOS uses the backslash (\) for paths and the forward slash (/) for command-line options.

What a mess! An undocumented DOS function, INT 21h Function 3701h, can help clean up this situation. This function changes the switch character; it is described in the appendix. This facility was documented for a brief time in the DOS user interface (the SWITCHAR= option in the DOS 2.0 CONFIG.SYS), but then it was literally undocumented.

This function can be incorporated in a tiny utility that sets the DOS SWITCHAR. Packages of UNIX utilities for DOS, such as the wonderful MKS Toolkit, include a similar utility:

```
/*
SWITCHAR.C -- uses undocumented DOS Function 3701h
switchar    changes DOS switch char to - and path char to /
switchar \  restores DOS switch char to / and path char to \
*/

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

main(int argc, char *argv[])
{
    int c = (argc > 1) ? argv[1][0] : '-';
#ifdef __TURBOC__
    _DL = c;
    _AX = 0x3701;
    geninterrupt(0x21);
    _AH = 0;   /* value returned in AX */
#else
    _asm {
        mov dl, c
        mov ax, 3701h
        int 21h
        xor ah, ah   ; value returned in AX
        }
#endif
}
```

After it is compiled with either Microsoft C 6.0 or Turbo C, SWITCHAR.EXE can be used (in DOS 2.x and 3.x) to make COMMAND.COM input a little more reasonable:

```
C:\UNDOC>dir -w /pharlap/*.exe
Invalid parameter

C:\UNDOC>switchar -

C:\UNDOC>dir -w /pharlap/*.exe

 Volume in drive C is RAMANUJAN
 Directory of  C:\PHARLAP
```

```
386LINK  EXE    CFIG386  EXE    386LIB   EXE    386DEBUG EXE    386ASM   EXE
RUN386   EXE
        6 File(s)   1261568 bytes free
```

Although DOS input is changed, notice that output isn't: the directory listing still uses the backslash. Also, note that some applications ignore SWITCHAR, insisting that you use backslashes for subdirectories.

Finally, note that most of COMMAND.COM in DOS 4.0 (including the DIR command) completely ignores the SWITCHAR. Setting SWITCHAR with Function 3701 is useful only when other programs, in particular COMMAND.COM, bother to call Function 3700 (Get SWITCHAR).

Distinguishing Internal and External Commands Every command interpreter implemented for any microcomputer has included at least some "internal" commands; most have also provided a means of executing "external" commands.

What distinguishes an "internal" from an "external" command is the location of the code that executes it. All internal commands are built into the command interpreter itself; external commands reside elsewhere. In some systems, including Tandy's TRSDOS among microprocessors and Honeywell's GCOS in the mainframe world, external commands have been stored in special library files. In MS-DOS, however, they are stored as individual program files.

CP/M contained only five internal commands. They were DIR, REN, TYPE, ERA, and SAVE. All other commands were external, stored as "COM" (for COMmand) files in a memory-image format. The first four of these were functionally the same as their MS-DOS descendants DIR, RENAME (REN), TYPE, and ERASE (also known as DEL). The other one provided the means for creating the necessary COM files: it would SAVE the specified number of 256-byte "pages" to a named file. To load a newly created program into RAM so that SAVE could do its thing, you had to use DDT.COM (certainly the best-named debugger ever).

The earliest versions of MS-DOS had only a bit more in the way of internal commands. The name ERA changed to ERASE, and DEL was added as a synonym. Similarly, REN was expanded to RENAME, with the older short form retained also. COPY, which under CP/M had been a function of the external utility called PIP (Peripheral Interchange Program, a concept inherited from DEC systems), moved into the command interpreter as an internal, and DATE, TIME, VER, and CLS were added. With each new version, as features were added to the system, additional internal commands came with them. The use of batch files alone gave birth to several internal commands to help make such files more useful.

By the time MS-DOS made it up to version 3.3, the list of internal commands had grown to 37. One undocumented command was added with version 4.0. The current list of commands (extracted directly from the internal table in the 4.01 version of COMMAND.COM) is as follows:

```
ERRORLEVEL                EXIST                DIR
CALL                      CHCP                 RENAME
REN                       ERASE                DEL
TYPE                      REM                  COPY
PAUSE                     DATE                 TIME
VER                       VOL                  CD
CHDIR                     MD                   MKDIR
RD                        RMDIR                BREAK
VERIFY                    SET                  PROMPT
PATH                      EXIT                 CTTY
ECHO                      GOTO                 SHIFT
IF                        FOR                  CLS
TRUENAME (undocumented)
```

The undocumented TRUENAME command displays the full physical path name of a file, given the file's name as its argument. Any SUBST replacement is translated from logical back to physical form, and any implicit path is made explicit. This command corresponds exactly to the INT 21h Function 60h, which is explained in greater detail in chapter 4 on the DOS file system.

For instance, if you issue the command "SUBST F: C:\ZAP\ZIP" and make F your current drive, the command "TRUENAME LZSS.C" displays the string "C:\ZAP\ZIP\LZSS.C".

Let's get back to the way the command interpreter determines whether it's dealing with an internal command or an external one: It performs a simple search of its internal command list. If the input command exactly matches any item in this list, it's internal and the corresponding internal command routine is executed. If it's not found, the interpreter treats it as a possible external command.

DOS 3.3 introduced a capability for extending the internal command list by way of TSRs that communicate with COMMAND.COM via a set of undocumented hooks. However, because this feature was not publicized, virtually no such extensions have yet been produced with the exception of the DOSSHELL capability that appeared first in DOS 4.0, and APPEND in DOS 3.3 (for which this functionality was probably created in the first place). We'll provide source code for an installable command later in this chapter when we examine the hooks that MS-DOS provides for command interpreters.

Note that the entire command line that was input to the interpreter is parsed before any attempt is made to locate the command. During parsing the command interpreter treats the "%" character as having special meaning, because it identifies references to command-line arguments when it is found in a batch file and it identifies environment variables. If followed by a character that is neither an argument identifier nor an environment variable's name, the "%" character normally is thrown away rather than being passed on to the command as part of the command line. This is true only when input is taken from a batch file (the "%" character is never thrown away if the input came through the keyboard buffer).

Sometimes the "%" character needs to be kept rather than thrown away. The classic case is the FOR command, with its "internal variable" reference, as in "FOR %f IN (*.c) DO type %f". That line will work perfectly from keyboard input, but if it is included in a batch file, both "%" characters will be dropped, and the command then generates a syntax error. To solve this problem, whenever COMMAND.COM (and its functional equivalent alternates) see a "%" character followed immediately by another identical character, it replaces them with a single "%", which passes on to the program. This action is equivalent to the C-language conventions regarding "\", which require that you use "\\" in any string where you want a single "\" to appear.

Finding and Executing Internal Commands Because the internal command list is searched first, it's a tricky matter to coax COMMAND.COM to run a program that has the same name as one of the internal commands. That is, if you name a program file "TYPE.COM", you'll find that the internal command "TYPE" takes its place when you try to execute the program via the command interpreter, although it runs your own program perfectly when invoked through the DOS EXEC function or via DEBUG.

When COMMAND.COM searches its command list for a possible internal command, it breaks off the command word at the first imbedded period after the command. Thus, to continue the example, if you entered "TYPE.COM" at the prompt, the ".COM" would be ignored, and the internal search would find a match to "TYPE," which would then be executed.

With more recent versions of DOS that permit you to specify a full path name for a command, you can solve this problem by specifying the full path to TYPE.COM, such as ".\TYPE" if the file is in the current directory. This does not, however, work with DOS 2.x, which is still used by many people.

With COMMAND.COM, the only general solution to this problem is to patch your copy of COMMAND.COM itself to change the name of the conflicting internal command. (The most recent versions provide another solution, however, which is explored later in this chapter.) At least some of the alternative command replacements attack the problem in two ways: by letting you selectively disable any internal command via a configuration option, and by a variation in the way the search is performed. The variation is simply that the input is not truncated at a period. If you enter "TYPE.COM", that is what will be searched for, and of course it will not be found because none of the internal commands have embedded periods.

Under any command interpreter, once an internal command is detected, it's normally executed immediately. The usual method of executing the command is by a call to the appropriate internal routine, followed by a loop back to the toplevel prompt code. Because all of this code is contained within the interpreter itself, no program swapping occurs, nor is any child process spawned for the majority of internal commands.

A few internal commands (notably the CALL batch file ability added to COMMAND.COM in DOS 3.3 and the HELP feature built into the alternate interpreter 4DOS.COM) do require that additional files be accessed. Such internal commands may fail if their additional files cannot be found. Except for such cases, however, internal commands run entirely within the currently loaded copy of the command interpreter and are not influenced by external events.

Before the internal command CALL was introduced, the external command COMMAND with the "/C" option switch was used to run one batch file from inside another, then return to the original file. This technique still works, but requires about 4KB additional RAM for the child copy of COMMAND.COM that is loaded, and may be significantly slower due to the additional disk accesses required.

Dispatching Appropriate Processes

Any name that is not matched in the internal command list is presumed to be an external command. The command interpreter searches for a file of that name, using the PATH to determine where to search. If such a file is located, it is loaded and run. If not, the error message "Bad command or file name" tells the operator that the input was faulty, and the command interpreter then returns to its toplevel "get input" procedures for a fresh command to interpret.

Locating and Loading External Commands To locate and execute an external command, the command interpreter first searches the current working directory for a file having a name identical to the command word and the extension ".COM". If this fails, the search is repeated using the extension ".EXE", and if this also fails, a third search is made with the extension ".BAT". Thus if three files named DO-ME.COM, DO-ME.EXE, and DO-ME.BAT all exist in the current directory, only the .COM file will be executed as an external command.

This "COM, EXE, BAT" sequence is established by code in the command interpreter, not by MS-DOS itself. It can be exploited in several ways that we'll examine a little later, but if you want to change it for any reason, you can do so by locating in your command interpreter the nine bytes that contain the characters "COMEXEBAT" and changing them as you desire. This can be done on the disk copy of the interpreter or in memory using DEBUG.

If all three searches fail, in the current working directory, the command interpreter looks for an environment variable named "PATH" and, if one exists, it takes the first path listed in that string (that is, all characters up to but not including the leftmost semicolon) and repeats the triple search in the directory specified by that path.

If unsuccessful, the interpreter moves on to the next pathspec in the PATH variable and repeats its actions. This continues until one of two things happens: a file is found that satisfies the search, or the PATH variable is exhausted without finding such a file.

If no file is found, the command interpreter issues a "Bad filename or command" error message and returns to the prompt. Note that this happens only after three searches have been made in each directory specified by PATH. If you have a large number of directories in your PATH, and each has a large number of files, the command interpreter may take a significant amount of time to conduct such a search.

Dealing with BAT Files When a file is found that satisfies the search criteria, the command interpreter's next action depends on whether the file found was a batch file (".BAT" extension). If so, the command interpreter sets appropriate flags to indicate to itself that it is processing a batch file rather than keyboard input, and stores enough data about the file to be able to find it again. The flag locations, and the amount of data saved, vary significantly from one version of DOS to the next.

The interpreter opens the batch file, reads its first line into the command input buffer, and replaces any indicated arguments with corresponding words from the original input line (which is retained in a separate buffer). It then closes the batch file and interprets and executes the modified batch-file line just as if it had been typed from the keyboard.

When that line is fully executed and control returns to the command interpreter, the flags tell the command interpreter to reopen the batch file, read the next line (actually, reads a minimum of 32 bytes) and execute it. This process continues until all lines of the batch file have been executed.

Note that the commands contained in the batch file can be either internal or external and that, since DOS 3.3, these commands can invoke additional subsidiary batch files in subroutine fashion via the CALL internal command. For these reasons, it's quite possible to invoke a batch file that never finishes executing, with the result that control never gets back to the original command interpreter's keyboard-input level. This is, in fact, the normal situation when you install a menu program or, in DOS 4.x, when you use the DOSSHELL facility.

Dealing with COM and EXE Files If the file found was not a batch file, the command interpreter uses the DOS EXEC function (INT 21h, Function 4B00h) to spawn the file's execution as a child process of the command interpreter, and nothing more happens in the interpreter until the child process terminates.

Notice that, with COMMAND.COM, it makes no difference whether a COM or EXE file was found; the distinction between the two types of executable files is made by the EXEC function based only on the first two bytes of the file (which in .EXE files have the "MZ" signature), and the actual file extension is ignored except to locate the file. This means that you could force a file that is really an EXE type to be found during the first search by changing its extension to COM.

An alternative to changing the file's extension is to create a "stub loader," which is a small file of the same name except for the extension COM. The stub loader then uses the EXEC function to spawn the original EXE file as a child of its own. This approach makes it possible to set up all sorts of special conditions before the program file is executed, then subsequently undo them, with minimal overhead.

One widespread use of this technique is in support of the third-party replacement video BIOS package UltraVision, from Personics. UltraVision permits its users to set up a wide variety of screen options, ranging up to 134x60. These formats are compatible with any other program that is "well behaved" in the sense

that it looks in the BIOS work area to determine what number of columns and rows are currently set. Unfortunately many popular programs don't bother to do so, because the ability to set "non-standard" formats is relatively recent. Among such "non-well-behaved" programs are both QuickC and Turbo C, together with many word processors.

To run programs that do not check the current number of columns and rows, it's necessary to set the screen format to the standard 80x25 dimensions. Personics' UltraVision generates a stub loader for any desired EXE-format program file. The loader first notes all pertinent information from the BIOS area and then resets to the 80x25 format. Next, it EXECs the specified program file, using its full name, including the EXE extension, and passing to it all the command-line arguments that the loader itself received. Upon return, the loader temporarily saves the exit code from the real program, restores the video set-up to the conditions it found at entry, and returns its child's exit code to DOS as though it were its own. The command interpreter always finds the COM-named loader first, so as a user you do nothing different.

As a result, by using the utility supplied with UltraVision to create a loader, you can make any program well-behaved in the video area, at the cost of less than a thousand bytes of overhead code.

Another use of the stub loader concept is built into Microsoft's new segmented-executable files, which are used in Windows, OS/2, and the European OEM multitasking DOS4. These new .EXE files have an "normal" old-style .EXE file at their head, followed by a new .EXE header with the "NE" signature. The old-style .EXE can be used to print a message such as "This program requires Microsoft Windows," or it can be used as a loader that, for example, goes ahead and runs Windows.

The Exitcode Concept At this point you may be wondering why a loader should preserve the exit code of a spawned process, or possibly even what an *exit code* is all about. Although it's documented (at least with regard to the method by which a program can supply one to its parent), the details are so spread out that some explanation is in order.

The idea that a process should return a result code to its parent followed directly from the concept that every process in a system is equivalent to a subroutine that is called by some higher-level process, all the way back to the primary bootstrap loader. This concept apparently originated at about the same time as

the idea of the operating system itself, long before microcomputing came upon the scene.

The term *exit code* became associated with this process result code during the development of the Multics multiprocessor system in the mid-1960s. It came to MS-DOS from Multics' descendant UNIX along with subdirectories and other Multics concepts that were grafted into MS-DOS at version 2.0.

Any process returns an exit code to its parent; the code can be controlled by the program if the process terminates via INT 21h functions 4Ch (Terminate with Exit Code) or 31h (Terminate But Stay Resident). If any other termination method is used (such as INT 21h Function 0), a default exit code value is generated by DOS itself.

The code can be retrieved, once and only once, at any time after the spawned process returns to the parent and before any subsequent process is spawned. The reason for this restriction is that DOS itself provides only one 16-bit word to hold the exit code, so that every process overwrites the code left there by its predecessor (or child), and the DOS function that retrieves the code (INT 21h Function 4Dh, Get Exit Code of Subprogram) zeroes out that storage location in the process of retrieving the code.

COMMAND.COM provides the internal "command" ERRORLEVEL, which is actually a function that can be evaluated by the internal IF command. ERRORLEVEL will retrieve the exit code of the most recent command and compare it to a specified value. All compatible command interpreters provide this command implemented in a functionally identical manner. When ERRORLEVEL is first called, it retrieves the exit code from DOS and stores it in another location inside the command interpreter. Then it sets a flag so that any subsequent call of ERRORLEVEL before an external command is executed will use that same value rather than attempting to retrieve it from DOS again.

The concept of exit code and ERRORLEVEL applies only to external commands; most internal commands have no effect at all on the ERRORLEVEL value, and because they do not spawn child processes, most commands do not affect the DOS exit code value. It's possible that some third-party command interpreters may, in future versions, extend the ERRORLEVEL idea to at least some internal commands; extending it to all commands (including those that test it) would negate the whole idea because it would make multiple-way decisions impossible.

The Hooks MS-DOS Provides

To facilitate the tasks that are common to all command interpreters, the designers of MS-DOS incorporated a number of "hooks" into the system. These include specific function calls that provide access to undocumented features, special dedicated interrupt vectors, and the entire Multiplex Interrupt concept. However, because these hooks were never officially documented, most of them are used less widely than they could be.

Dedicated Interrupt Vectors Two interrupt vectors (INT 2Eh and INT 2Fh) are reserved by MS-DOS for support of the command interpreter. Because it is implemented within the command interpreter itself, details of INT 2Eh are deferred until the following section.

The Multiplex Interrupt concept evolved rather slowly as DOS advanced, and it may have changed again by the time this description sees print. Originally, INT 2Fh was only a method by which PRINT.COM (the first official TSR program) could communicate with DOS while it was not the current process. In DOS 2.x, the only built-in support for INT 2Fh was that in PRINT.COM. However, the third-party developers of TSRs rapidly discovered its existence and began using it for their own needs.

By the time DOS had advanced to version 3.0, the idea of the Multiplex Interrupt had become well established and a number of other functions had been assigned function codes. More importantly, a set of ground rules for use of this interrupt had been established and publicized. As a result, it became possible for users to hook into this vector even though the details of much of its action remained undocumented.

The major ground rule was that every function supported by INT 2Fh should be assigned a function number, which is passed to the interrupt service routine in the AH register.

Any interrupt service routine (ISR) that hooks the INT 2Fh vector is expected to preserve a chain to the previous routine, and when invoked, it should look for its own function number in AH and pass the request on if no match exists (that is, INT 2FH service routines form a chain, each routine looking only for requests addressed to it).

The AL register was similarly reserved for subfunctions, and these codes were allowed to vary from one routine to another, except that subfunction 00 was globally defined to be "Are you there?" (install check). Any ISR, upon receiving a

subfunction 00 message for its function number (AH value), is expected to set AL to 0FFh and return. Because the default service for this interrupt is the single opcode IRET, any request that fails to find a match will return with AL still all zeroes; this indicates that the requested service is not installed.

The major violators of these ground rules were, not surprisingly, some of the programs supplied along with DOS! The most dangerous of these violations is the use of Function 13h, in DOS 3.3 and up, to modify the address of the disk interrupt handler for DRIVER.SYS and the Installable File System. This routine does not obey the "Are you there" rules, but rather changes the address to whatever is contained in the registers. This, of course, can be disastrous to all disk I/O from that point on. Even worse, it returns the previous address in the registers, and has been used by at least two virus programs to obtain undetected access to the disk services.

Less serious problems have been caused by changes in the assigned function codes between versions. For example, Function 15h is documented as being the CD-ROM extension service code (MSCDEX), but it is also used in DOS 4.0 and up to verify that GRAPHICS.COM has been installed.

Tabulation of Multiplex Interrupts Functions Only a few of the current crop of Multiplex Interrupts are directly involved with command interpreter support. For a more complete list of these functions, refer to the Appendix.

The services in the following list are those called by the COMMAND.COM supplied with MS-DOS 4.01; earlier versions and other command interpreters may not use them:

Function 05h, CRITICAL ERROR HANDLER, available since DOS 3. This function, called after INT 21h Function 59h has expanded an error code into its locus and recommended action, converts those codes into readable text, in the language determined by the system configuration when DOS was installed. This is the method by which the cryptic error messages given by older DOS versions were expanded into more informative ones at the introduction of version 4.0.

Function 12h, DOS INTERNAL SERVICES, available since DOS 3. This function provides access to a wide variety of DOS internal functions. Many of them are useless from outside the DOS kernel itself, because they depend on both DS and SS being set to the DOS kernel segment, but some of them are of general use even though they duplicate other DOS functions. COMMAND.COM uses several of these. One of the functions used most frequently by COMMAND.COM, Func-

tion 122Eh, establishes the pointers used by the critical error handler translation function described in the previous paragraph.

Although one of the Function 12h subfunctions will give you the segment address of the DOS kernel, that's not really adequate for making use of this group of routines. Many of them temporarily change the value in DS, and rather than PUSHing the original value and POPping it back, they "restore" it when required by using a "PUSH SS" followed by "POP DS". This requires that not only DS, but also SS, be set to the DOS segment, and if that is done, then SP must be set to one of the (several) stacks used by DOS. Unfortunately, even if you could locate the right stack easily, using it would then prevent you from making any use of DOS itself! That's why so many functions in the INT 2Fh Function 12h group are tagged as being callable only "from within a DOS function call" in the appendix.

Function 19h, SHELLB.COM, available since DOS 4. This function is defined only in DOS 4 and up; it provides run-time interfacing between the DOSSHELL batch file and the two executable programs, SHELLB.COM and SHELLC.EXE, that implement the full-screen interface. These do not replace COMMAND.COM, but rather run as a pair of TSRs as child processes. SHELLB installs the INT 2Fh handler for this function, and SHELLC takes care of such mundane details as the CRT display, dispatching necessary child processes, and so on. This function, together with the following one, become one of the alternate input sources available to COMMAND.COM when the DOSSHELL facility is in use.

Installable Commands Function AEh, INSTALLABLE COMMAND, available since DOS 3.3. This function connects what would otherwise be an external command into the internal command list of COMMAND.COM. Both the command code itself and the handler for this function must be installed as a single, non-pop-up TSR. This capability was introduced with DOS 3.3, but the only well-known example of its use is the DOS 4 DOSSHELL facility. In DOSSHELL, the SHELLC.EXE file contains several installable internal commands and the SHELLB.COM file provides the INT 2Fh handler.

You can use this undocumented interrupt to install your own commands, but they will work only with DOS 3.3 and above.

The basic principle on which this function operates is "Don't call us, we'll call you." COMMAND.COM issues calls to this function at four places while parsing the input line. Two of these calls are to subfunction Function AE00h, to determine if the command on the input line is a valid installed command; the other two are called only if the first calls return FFh in the AL register, indicating that the com-

mand is indeed valid. The second calls are both to subfunction Function AE01h, which executes the command.

When Subfunction AE00h is called, both the DS and ES registers point to COMMAND.COM's transient area. The DX register is set to 0FFFFh for unknown reasons, the BX register points to a buffer that contains two count bytes followed by an exact copy of the input line, and the SI register points to a different buffer that contains only the command word from the line, converted to uppercase and preceded by its character count. Your code must verify the content of this second buffer is an exact match for the name of your installed command; if not, it should chain the interrupt on up the line in case some other installed command is being called. If the second buffer does match your installed routine, your routine should change the AL register to FFh and return to the caller.

When Subfunction AE01h is called to execute the command, the ES, DS, BX, DX, and SI registers contain exactly the same data as for the previous subfunction, although only DX and SI are again explicitly loaded with the values prior to the call. The BX register has been preserved through a couple of subroutine calls, but it may not always retain the pointer to the full input line buffer.

Therefore, if your command accepts arguments on the input line, it's recommended that those arguments be copied into a local buffer before the routine returns from the AE00h call. After executing the actual command within the AE01h call, your code should zero out the count byte in the command buffer at DS:SI. This signals COMMAND.COM not to attempt to execute the command itself.

These calls are made before the internal command list is checked, which means that you can install a command that has the same name as one of the internal commands, and your command will replace the original one. Because your own command can be a do-nothing thing, this offers an elegant way to disable the DEL and ERASE commands on a system that must be accessible to the general public.

The following C program, INSTCMD.C, illustrates all the points of dealing with the installable command interrupt service. Note that this program does not install as a TSR, but instead shells to a child copy of COMMAND.COM to avoid problems in deinstalling the added command.

```
/*
INSTCMD.C

The "Installable Command" function is not called by a program that
```

wants to extend COMMAND.COM's repertoire. Instead, you hook the function and wait for COMMAND.COM to call you ("don't call us, we'll call you"). Function AE00h lets you tell COMMAND.COM whether you want to handle the command, and function AE01h is where you actually handle it (similar to device driver division of labor between Strategy and Interrupt).

Note that AE01h is called with only the name of the command: not with any arguments. Therefore, arguments must be saved away in AE00h. Yuk!

Furthermore, while redirection is handled in the normal way in AE01, in AE00 we get the entire command string, including any redirection or piping. Therefore, these must be stripped off before saving away the args during AE00 processing.

Problem with following AE00 and AE01 handlers: they should chain to previous handler. For example, INTRSPY program won't see AE00 and AE01 once INSTCMD is installed.

The sample COMMAND.COM extension used here is FULLNAME, based on the undocumented TRUENAME command in DOS 4.x. We simply run undocumented Function 60h in order to provide FULLNAME. Actually, not quite so simple, since Function 60h doesn't like leading or trailing spaces. These are handled inside function fullname().

The following INTRSPY script was helpful in debugging 2FAE:

```
; INSTCMD.SCR
structure cmdline fields
    max (byte)
    text (byte,string,64)

intercept 2fh
    function 0aeh
        subfunction 00h
            on_entry
                if (dx == 0FFFFh)
                    output "AE00"
                    output (DS:BX->cmdline)
        subfunction 01h
            on_entry
                if (dx == 0FFFFh)
                    output "AE01"
                    output (DS:SI->byte,string,64)

requires Microsoft C 6.0+, or Quick C 2.0+
    cl -qc instcmd.c
```

```
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <conio.h>
#include <dos.h>

#pragma pack(1)

typedef struct {
    unsigned es,ds,di,si,bp,sp,bx,dx,cx,ax;
    unsigned ip,cs,flags;
    } REG_PARAMS;

typedef unsigned char BYTE;

typedef struct {
    BYTE len;
    BYTE txt[1];
    } STRING;

typedef struct {
    BYTE max;
    STRING s;
    } CMDLINE;

void interrupt far handler_2f(REG_PARAMS r);

void (interrupt far *old)();

void fail(char *s) { puts(s); exit(1); }

main(void)
{
    /* hook INT 2F */
    old = _dos_getvect(0x2f);
    _dos_setvect(0x2f, handler_2f);

    puts("This demo of installable commands isn't a TSR.");
    puts("Instead, it just creates a subshell from which you can EXIT");
    puts("when done. In the subshell, one new command has been added:");
    puts("FULLNAME [filename].");

    system (getenv("COMSPEC"));
```

```
    /* unhook INT 2F */
    _dos_setvect(0x2f, old);
}

char far *fullname(char far *s, char far *d)
{
    char far *s2;

    /* INT 21h AH=60h doesn't like leading or trailing blanks */
    while (isspace(*s))
        s++;
    s2 = s;
    while (*s2) s2++;
    s2--;
    while (isspace(*s2))
        *s2-- = 0;

    _asm {
        push di
        push si
        les di, d
        lds si, s
        mov ah, 60h
        int 21h
        pop si
        pop di
        jc error
        }

    return d;
error:
    return (char far *) 0;
}

void fcputs(char far *s)
{
    /* can't use stdio (e.g., putchar) inside 2FAE01 handler? */
    while (*s)
        putch(*s++);
    putch(0x0d); putch(0x0a);
}

char buf[128];      /* not reentrant */
char args[128];

#define CMD_LEN     8
```

```
void interrupt far handler_2f(REG_PARAMS r)
{
    if ((r.ax == 0xAE00) && (r.dx == 0xFFFF))
    {
        CMDLINE far *cmdline;
        int len;
        FP_SEG(cmdline) = r.ds;
        FP_OFF(cmdline) = r.bx;
        len = min(CMD_LEN, cmdline->s.len);
        if ((_fmemicmp(cmdline->s.txt, "fullname", len) == 0) ||
            (_fmemicmp(cmdline->s.txt, "FULLNAME", len) == 0))
        {
            char far *redir;
            int argslen = cmdline->s.len - CMD_LEN;
            _fmemcpy(args, cmdline->s.txt + CMD_LEN, argslen);
            args[argslen] = 0;
            /* yuk! we have to get rid of redirection ourselves! */
            /* it will still take effect in AE01 */
            /* the following is not really correct, but okay for now */
            if (redir = _fstrrchr(args, '>'))
                *redir = 0;
            if (redir = _fstrrchr(args, '<'))
                *redir = 0;
            if (redir = _fstrrchr(args, '|'))
                *redir = 0;
            r.ax = 0xAEFF;      /* we will handle this one */
        }
    }
    else if ((r.ax == 0xAE01) && (r.dx == 0xFFFF))
    {
        STRING far *s;
        int len;
        FP_SEG(s) = r.ds;
        FP_OFF(s) = r.si;
        len = min(CMD_LEN, s->len);
        if ((_fmemicmp(s->txt, "fullname", len) == 0) ||
            (_fmemicmp(s->txt, "FULLNAME", len) == 0))
        {
            char far *d;
            if (! *args)
                d = "syntax: fullname [filename]";
            else if ((d = fullname(args, buf)) == 0)
                d = "invalid filename";
            fcputs(d);
            s->len = 0; /* we handled it; COMMAND.COM shouldn't */
        }
```

```
    }
    else
        _chain_intr(old);
}
```

Only the Microsoft compilers are capable of dealing with INSTCMD.C, because Turbo C's libraries do not provide an equivalent to the _chain_intr() function that is used to pass the request up the Multiplex Interrupt handler chain.

If you run INSTCMD.C, you will first see the four-line explanation of what the program does, followed by the banner line as the child copy of COMMAND.COM initializes itself. From that point until you type EXIT to return to your parent process, you will have available the added command "FULLNAME." This command is functionally identical to the undocumented "TRUENAME" command introduced with DOS 4.0 and described elsewhere in this chapter.

Notice that in INSTCMD.C, we copy the arguments supplied on the input line into a buffer when servicing subfunction AE00 and then use them when executing the command subsequently. Both arguments are in the function handler_2f().

Finally, notice how we have the opportunity to display a help message—something which COMMAND.COM itself could do for internal commands, but doesn't. This, combined with the fact that we can install front-ends, in addition to replacements, for existing internal commands (simply by returning AL=0 after handling the command), means that the installable-command facility can also be used to create DOS help systems.

TSHELL, a Simple Command Interpreter

The basic ideas behind a command interpreter are extremely simple. What makes them seem complicated in practice is the need to handle all possible circumstances, with minimum disruption of system operation. To show just how simple a command interpreter can be and still function, here's TSHELL.C, a very tiny shell that you can actually install as the primary shell of your system:

```
/*********************************************************
 *  TSHELL.C - Demonstration tiny command interpreter     *
 *  Jim Kyle, July 10, 1990                               *
 *                                                        *
 *  Intended only to show basic principles; not for use   *
 *  with DOS versions prior to 3.1 (EXEC function of such *
 *  versions does not preserve stack registers).          *
```

```
 *                                                          *
 *  For Turbo C only due to pseudovariable usage.           *
 *                                                          *
 *      tcc -mt -c tshell                                   *
 *      tlink /t /c c0t+tshell,tshell,,cs.lib               *
 *                                                          *
 ************************************************************/
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include <dir.h>

char cmdbuf[128];
char *cmdlst[] = {"DIR","RUN"};
int i;

void do_dir( void )     /* reports files in cur dir     */
{ struct ffblk wkarea;
  int endir;

  if (strlen(cmdbuf) < 5)       /* default to all files */
    strcpy( cmdbuf+4, "*.*" );
  puts("\n    Files and sizes\n");
  endir = findfirst( cmdbuf+4, &wkarea, 0 );
  while ( !endir )
    { printf("%-13s   %8ld\n",
             wkarea.ff_name, wkarea.ff_fsize );
      endir = findnext( &wkarea );
    }
  putchar( '\n' );
}

void do_run( void )     /* caution, safe only for DOS3+ */
{ struct {
    unsigned eseg, clo, cls;
    long     fcb1, fcb2;
  } parms;
  cmdbvf[0] = strlen (cmdbuf+1);
  cmdbuf[1] = '/';
  cmdbuf[2] = 'C';
  cmdbuf[3] = ' ';
  parms.eseg = 0;
  parms.clo = (unsigned) cmdbuf;
  parms.cls = _DS;
  parms.fcb1 = parms.fcb2 = 0L;
  _ES = _SS;
  _BX = (unsigned) &parms;
```

```
  _DX = (unsigned) "C:\\COMMAND.COM";
  _AX = 0x4B00;
  geninterrupt( 0x21 );
}

void main( void )
{ puts( "  TINY SHELL DEMONSTRATOR\n" );
  puts( "Copyright 1990  by Jim Kyle\n" );
  puts( "  Commands: DIR, RUN only\n" );
  for( ; ; )
    { printf("tinyshell> ");
      gets(cmdbuf);
      for(i=0; i<2; i++)
        if(strnicmp(cmdlst[i],cmdbuf,strlen(cmdlst[i]))==0)
          break;
      switch(i)
        { case 0: do_dir();
              break;
          case 1: do_run();
              break;
          default: puts("Unknown command!!!\n");
        }
    }
}
```

Of course, this just shows the bare rudiments of a DOS command interpreter. A real one would have to handle INT 23h (Ctrl-C) and INT 24h (Critical Error) at the very minimum. It also would be important to take over undocumented INT 2Eh, discussed later in this chapter, even if only to point it at an IRET instruction.

Unlike the other programs in this chapter, TSHELL.C is written to be compiled only with Turbo C (or JPI TopSpeed C, which also supports geninterrupt() and register pseudovariables). The interface to the DOS EXEC function in this program cannot rely on any environment being established (remember, COMMAND.COM isn't running!), and the library functions of both Microsoft's and Borland's products do use the environment. Thus, it was necessary to drop to assembly language for the interface, and the Turbo C pseudovariables were used for the same reasons that DEVLOD.C in chapter 3 also used the Borland product.

Note that TSHELL is not intended to be useful; its sole purpose is to present the skeleton of a command interpreter. It recognizes only two commands, DIR and RUN, and the RUN command actually launches COMMAND.COM as a child process in order to make the conventional DOS command set available.

Without this capability, it's much more difficult to get TSHELL out of your system once you've tried it out!

Once you have TSHELL.COM, add a line such the following to your CONFIG.SYS file (temporarily REM out any existing SHELL= statement):

```
SHELL=C:\TSHELL.COM
```

Note that you must use the SHELL= command in CONFIG.SYS to install a different command interpreter. Many people are under the impression that it can also be installed by changing the COMSPEC= variable in the environment, but the environment variable is used only by the existing command interpreter to reload itself when necessary. Until a command interpreter is loaded and initialized, neither the COMSPEC variable nor the master environment block itself exists!

Now reboot the system. You'll see that the prompt is now "tshell>" rather than anything you may be used to seeing. You'll note, also, that AUTOEXEC.BAT did not run, and none of your TSR programs have been installed.

If you type DIR, you'll get a list of programs in the current directory, but no subdirectories will be shown, nor will the amount of available space. Any other conventional command will produce only an error message, because TSHELL does not recognize it.

When you type RUN, either with or without any arguments, you can expect to see two error messages complaining about a bad search directory, followed by the conventional COMMAND.COM banner as a child copy of COMMAND.COM goes into action. From this point, you have all normal commands available to you, which lets you change CONFIG.SYS back to get rid of the TSHELL line.

The reasons for the two error messages are not fully known; our best guess is that they result from TSHELL's attempt to spawn a child copy of COMMAND.COM when no primary copy was loaded, but they could also result from the fact that no master environment is created by TSHELL. You can verify the absence of a master environment by executing the EPTST.EXE program presented later in this chapter from the child COMMAND.COM spawned by TSHELL's RUN; it will show no master environment, and SET will show that "PATH=" is blank in the current copy.

The key point about TSHELL, though, is the skeletal structure of a command interpreter provided in main(): an endless for (;;) loop which prints a prompt (a

more complete implementation would interpret and print getenv("PROMPT")), gets commands, interprets them, and executes them.

How COMMAND.COM Works

This section is based on disassembly of several versions of COMMAND.COM; its major emphasis is on those points not already adequately covered in official system documentation. This section also defines such terms as *master environment* and *primary shell.*

Although COMMAND.COM is not the only possible command interpreter (and later in this chapter we look briefly at some alternatives), it is the one most used with MS-DOS, because it comes as part of the system package. To many users, it *is* the operating system, because the real system files are hidden from view. (Actually, in DOS 2, COMMAND.COM was an essential part of the operating system, because the DOS EXEC Function 4Bh was actually contained in the resident portion of COMMAND.COM rather than in the DOS kernel. By the time DOS 3.0 was released, however, EXEC had been moved to its proper location, and the operating system itself became independent of COMMAND.COM.)

The process of creating the primary shell actually begins when hidden file IO.SYS in MS-DOS (or IBMBIO.COM in PC-DOS) is loaded and its initialization code takes over. After a bit of preliminary calculation, this code moves the part of itself that has not yet been executed up to the topmost memory area physically present in the 640KB "DOS memory," much as the DEVLOD program in chapter 3 moved itself during execution.

From that vantage point, out of harm's way, the IO.SYS code then installs the rest of the DOS kernel, making it possible for the primary shell to use all the DOS services when it loads for the first time.

The "primary shell" defined in the CONFIG.SYS file (which defaults to COMMAND.COM if no SHELL= line occurs in CONFIG.SYS) is loaded as essentially the final step of initialization by IO.SYS. After building all required data structures for use by DOS, the IO.SYS initialization procedures use INT 21h Function 4B00h, to load and execute the primary shell program.

One of the parameters passed to DOS with this request is the address of the associated environment block. For the primary shell, a "master environment block" is assigned just above the DOS kernel area, by code in the COMMAND.COM initialization routines. The size of this block defaults to 160 bytes, but it can

be expanded by means of the "/E:" option switch added to a "SHELL=COMMAND.COM /P" line in CONFIG.SYS.

For primary shells other than COMMAND.COM, the exact method may differ, but all provide methods for tailoring the size of the master environment to whatever you need.

If control ever returns to the IO.SYS initialization code from that primary shell, a fatal error has occurred (such as the stack becoming corrupted or critical system code being modified by accident) and continued operation is impossible. Therefore the call to INT 21h EXEC is followed by a "dynamic halt" (JMP $), which locks the system solid. Only pressing the reset button (if present) or powering down and then back up can return the system to operation.

The only time that control can return to this dynamic halt code is while the primary shell is loaded. The most likely reasons for such immediate return would be failure to locate the shell or inadequate memory to load it.

Failure to find the program is usually the result of a spelling error in the CONFIG.SYS file or a move of the program to some directory other than that specified in CONFIG.SYS. If this happens, you'll need to have a bootable floppy handy to bring the system back up so that you can correct the errors in CONFIG.SYS.

As soon as the primary shell begins execution, the area at the top of DOS RAM from which IO.SYS called the shell becomes available for reuse. Subsequent operations while initializing the primary shell normally overwrite the EXEC call and the JMP $ which follows it.

The Division Points

When it is first loaded by the initialization code in IO.SYS, COMMAND.COM divides itself into three parts. One part stays where it was initially loaded, in low memory, just below the area where most user programs run. Another part moves to the highest available area within the 640KB DOS RAM limit. The middle portion is discarded after it finishes setting things up. Official DOS manuals tell us that much, but very little else; here's more of the story.

Resident, Initial, and Transient Portions The three parts into which COMMAND.COM divides itself are known as the *resident*, *transient*, and *initialization* portions.

The resident portion contains the ISR for INT 2Eh, and in some older versions of DOS it also contains some of the INT 21h service routines (including, as mentioned earlier, the EXEC function). In addition to these interrupt handlers,

the resident portion performs necessary clean-up when a terminated process returns control to the resident portion. This portion also contains the permanent data storage associated with the command interpreter (such as the pointers to the transient portion).

The transient portion is needed only for internal commands and is made available for use by any external command that needs the space. This area contains the actual input buffer, the code that interprets commands, the internal command list, and all of the code for executing internal commands. It does not contain code for dispatching external commands, however; if a command is deemed external, control transfers back to the resident portion. This assures that when the external command terminates, its return address (stored by DOS in processing the EXEC function) will still be valid.

Next is the initialization portion. Each time COMMAND.COM is loaded, at least some of this code is executed to verify that the version levels of COMMAND.COM and the resident DOS kernel match exactly, and to parse any arguments passed to the program. Other actions depend on whether the "/P" option switch was one of the arguments passed. If this switch is absent, initialization merely shrinks the RAM allocation and then goes directly to the input prompting routines to accept input.

If, however, the switch is present (as it is when the primary shell is being installed by IO.SYS), initialization calculates and stores the starting address for the transient portion, moves the transient portion into place at the top of RAM, calculates and stores its checksum for future use in reloading, and installs the interrupt handler for INT 2Eh. It then checks for the existence of a file named "AUTOEXEC.BAT" in the current working directory. If one exists, the initialization portion sets flags that direct the input routines to process it before looking for keyboard input.

In either case, this portion of COMMAND.COM next shrinks the RAM allocation to be just adequate to cover the resident portion (if this is not the primary shell, "resident portion" refers to this copy and not to the primary copy itself) and transfers control to the command input prompting routine. This routine issues the prompt, waits for input, and dispatches it. But the initialization portion, as such, ceased to exist when the RAM allocation was reduced.

Where These Portions Are Loaded The preceding descriptions show you how COMMAND.COM splits itself up for action, but for a full appreciation of what goes on you need to spend a few minutes at the keyboard examining exactly how

the various parts wind up in your own system. The major tool for doing this will be the standard DOS debugging tool, DEBUG.

To locate the resident portion of COMMAND.COM in RAM, use the INT 2Eh vector to find the segment address for the primary shell, as follows:

```
mov     ax,352Eh        ; get vector for INT 2Eh
int     21h             ; into ES:BX
```

A little later in this chapter, this technique is discussed in more detail, including why it works, when it doesn't work, and how it compares to other methods.

Once you have pinned down the resident portion, all sorts of possibilities beckon. Later in this chapter, this portion is used to build an editor for the master environment. And we'll use the resident portion's own data, right now, to find the transient portion tucked away in the upper reaches of DOS memory.

To locate the transient portion, use DEBUG, find the resident part and then use the command "S ES:0 Lfff0 59 4E" to search for the string "YN". This string should be found twice, but the second copy is just the one that DEBUG uses to compare against in the input buffer.

Display the first area found. The string should also contain the letters "ARIF," but not necessarily in that sequence; these are the acceptable responses to the "Abort, Retry, Ignore, Fail" error message from INT 24h, though "F" will be missing for DOS versions prior to 3.3. Look some 80–90 bytes past the string for a four-byte far pointer in which the first two bytes are "2C 01"; the next two bytes will be the segment address of the transient portion.

These four bytes are the far pointer that COMMAND.COM uses to transfer control to the transient portion upon return from a process. In IBM's DOS 4.01 with a full 640KB system, the pointer value is 9929:012C, but the segment will vary from version to version, as will the pointer's location. The offset has been the same in all versions since 3.1.

It's a lost cause to try to view the initialization code of COMMAND.COM in RAM, because it is overwritten immediately by the first process spawned. To view this code, you have to use DEBUG (or a more powerful disassembly tool) and look at the file itself.

Using DEBUG, the way to do it is with the command "DEBUG COMMAND.COM", which will get you the usual "-" prompt from DEBUG. Because it's a normal COM file, you can use either the P or the T command to jump to the first

byte of the initialization code; you can then use the U command (for unassemble) to view the code itself.

For serious study, you need to take notes of the addresses, paying special attention to the start and stop addresses. You can then use Q to quit DEBUG and create a script file consisting of the P, followed by the sequence of U <from> <to> commands, and terminated by a Q. This script file can then be fed to DEBUG in another session by using the command "DEBUG COMMAND.COM <INP.SCR >OUT.DAT". This assumes that you have named the script file INP.SCR; the command uses COMMAND.COM's redirection features to supply DEBUG the inputs from the script and to save all output in the newly created file OUT.DAT. You can then study OUT.DAT at leisure to learn more about how COMMAND.COM installs itself.

Using the Environment

This section discusses the Environment as implemented by MS-DOS and used by COMMAND.COM. Although the concept itself is firmly based at the operating-system level, the details of its usage are left to the command interpreter, with the result that this discussion may not apply fully if an alternate interpreter is in use.

How COMMAND.COM Uses the Environment The concept of an "environment" for each process came to MS-DOS from UNIX but was greatly simplified in the transfer. As currently implemented in MS-DOS, the environment consists of a paragraph-aligned block of space that may be up to 32,767 bytes in length (although in practice it is always much smaller). This block contains a collection of "environment variables," each of which consists of a variable name followed by the variable's data. Both the name and the data are stored as an ASCIIZ text string, with an equal sign ("=") separating the name from the data.

The link between the process and its environment is provided by DOS, which plugs the segment address of the environment into the word at offset 2Ch of the PSP when dispatching the process.

The program that does the dispatching is responsible for allocating the environment space and defining all the variables contained in it, with one exception. If the segment address passed to the DOS EXEC function is 0h, then DOS itself will allocate just enough space to contain a copy of each variable in the parent's environment and will do the copying automatically. The sole (and undocumented) exception to this rule is that the environment space used initially by the

primary shell must be set up explicitly by the primary shell itself when it initializes. Otherwise, no master copy of the environment will exist.

COMMAND.COM uses several predefined environment variables to control its actions. One, named COMSPEC, provides the drive, path, and file specification that is used each time COMMAND.COM must reload its transient portion. Another, PATH, lists the drives and paths to be searched for possible external command filenames. Still a third, PROMPT, stores the string of characters that COMMAND.COM uses to prompt for user input.

Both PATH and PROMPT have separate internal commands that can be used to modify their content. However, any environment variable can be modified by the internal command SET; if the variable does not exist, it is created in the environment, provided that enough space exists for it. From the primary command shell, the SET command operates on the master environment.

The predefined environment variables (PATH and COMSPEC only) are created in the master environment (that for the primary shell) by the primary shell's own initialization code. The size of the master environment is also determined at this time: it is 160 bytes by default, but can be altered by the "/e:nnn" option in the SHELL=COMMAND.COM line in CONFIG.SYS (alternate shells use different syntax but have the same capability). Once the primary shell has been loaded, its environment space allocation cannot be increased.

When COMMAND.COM dispatches an external command, it simply passes the 0h code to DOS, thus generating an exact copy of the master environment for use by the spawned process. Because it is a copy and not the original, any changes made to it by the process will not be reflected in the master environment itself.

Although this protects the master environment from being altered accidentally, it makes it difficult to alter it intentionally except by using the SET command from the primary shell command line prompt, which is not always convenient.

Locating the Environment Before you can make any use of the information stored in the environment area, you must first find it. DOS stores the segment address of the environment area for each process in the word at offset 2Ch in the PSP, but if you need access to the master environment, you must locate the PSP for the primary shell.

The following assembly language package (ENVPKG.ASM) contains three routines designed to support small-model C programs. The first two, curenvp() and mstenvp(), locate the current and the master environment areas, respectively,

and return far pointers to the first byte. The third, envsiz(), requires as input a pointer such as the one returned by the first two, and returns the size of the area in paragraphs.

```
;ENVPKG.ASM - Jim Kyle - July 1990

.model small,c

.data
; assumes being used from C with _psp global variable
        extrn   _psp:word

.code

curenvp proc

        public  curenvp
; char far * curenvp( void );

        mov     ax,_psp         ; get PSP seg
        mov     es,ax
        mov     dx,es:[002Ch]   ; get env address
        xor     ax,ax           ; offset is zero
        ret

curenvp endp

mstenvp proc

        public  mstenvp
; char far * mstenvp( void );

        mov     ax,352Eh        ; get INT2E vector
        int     21h             ; (master segment)
        mov     dx,es:[002Ch]   ; get env address
        xor     ax,ax           ; offset is zero
        ret

mstenvp endp

envsiz  proc    oenv:word, senv:word

        public  envsiz
; short envsiz( char far * vptr);

        mov     ax,senv         ; get segment of env
```

```
        dec     ax              ; back up to MCB
        mov     es,ax
        mov     ax,es:[0003h]   ; get size in grafs
        ret

envsiz  endp

        end
```

The lines following each of the "public" directives are sample prototype declarations to be copied into any C program that uses ENVPKG. To use these routines, first assemble the program into an OBJ file as follows:

```
MASM /mx ENVPKG;
TASM /mx /jMASM51 ENVPKG;
```

The "/mx" option switch, for both assemblers, forces procedure names to remain in lower-case so that the OBJ file can be linked with your C programs. The operation of curenvp() and mstenvp() also relies on the fact that C compilers for the PC return four-byte far pointers in the DX:AX register pair.

The curenvp() routine assumes an external global variable called _psp: this is provided in all C compilers for the PC, though some compilers declare _psp in DOS.H and others in STDLIB.H.

The envsiz() routine is based on the fact that every environment block starts at an offset of zero and is preceded by a Memory Control Block that contains the size of the block in paragraphs (see chapter 3).

Thus, when you pass a far pointer to the environment block to envsiz(), it decrements the segment by 1 to address its associated MCB and then retrieves the size from offset 3 in that segment.

Note that the returned value is always in paragraphs. If a byte size is needed, the returned value must be multiplied by 16 (or shifted left 4 places, which is faster).

This short C program, EPTST.C, illustrates use of the ENVPKG routines:

```
/*********************************************************
 *  EPTST.C - Tests environment-access object modules     *
 *  Jim Kyle, July 1, 1990                                *
 *********************************************************/
#include <stdio.h>
```

```
char far * curenvp( void );   /* prototype declarations   */
char far * mstenvp( void );
short envsiz( char far * vptr);

void main (void)
{ char far *mine;
  char far *master;

  puts("\nEnvironment locations are:");
  mine = curenvp();
  master = mstenvp();
  printf("Current environment is at %Fp, size: %i bytes\n",
          mine, envsiz(mine)<<4 );
  printf(" Master environment is at %Fp, size: %i bytes\n",
          master, envsiz(master)<<4 );
}
```

You can compile this program with either QuickC or Turbo C. In either case, be sure to compile using the small memory model, or alter your ASM file's ".model" statement to reflect the memory model that you will use.

From the command line prompt, you can compile with either of these commands:

```
tcc -ms eptst.c envpkg.obj
qcl /AS eptst.c envpkg.obj
```

Typical output from EPTST, when run at the primary shell's command prompt, follows:

```
Environment locations are:
Current environment is at 1868:0000, size: 256 bytes
Master environment is at 11D7:0000, size: 512 bytes
```

This shows how the current working copy of the environment has been trimmed to a size just adequate to hold the defined variable strings and the program's pathspec. Don't expect your own results to be identical to this example; the size of both the master and the current environments depends on what information you have stored in each.

Other Ways of Locating the Environment The preceding discussion has a problem: the interrupt vector for INT 2Eh *doesn't* always point to the primary copy of the command interpreter! For example, using David Maxey's INTRSPY program

from chapter 8, it is trivial to take over INT 2Eh for the purpose of seeing who calls it. If mstenvp() comes along while INTRSPY is in control of INT 2Eh, it will think that INTRSPY is the primary command interpreter, even though INTRSPY has simply hooked INT 2Eh for diagnostic purposes and is just chaining the interrupt to the previous owner which, presumably, *is* the primary command interpreter.

Of course, this is unlikely to happen out in the field: after all, nothing but a command interpreter really takes over INT 2Eh, right? Still, it's too easy to fool mstenvp() in its current form.

What other ways are there of locating the master environment?

There are plenty of other techniques, *all* of which require undocumented DOS:

One technique avoids the issue of *locating* the master environment entirely: it issues SET commands using INT 2Eh. The ability to call COMMAND.COM via INT 2Eh is discussed later in this chapter. Note that mstenvp() does *not* call INT 2Eh: it merely uses its interrupt vector in hopes of finding the primary command interpreter. This technique is repeated in mstenvp1(), in MSTENVP.C below.

An entirely different technique, used by the .BAT compilers discussed earlier in this chapter, walks the MCB chain looking for the environment segment belonging to the primary command interpreter. Chapter 3 documented how to walk the MCB chain. The key here is to take the *first* environment you find in the chain. How do you find the first environment? Don't look for certain ASCII characters to see if an MCB corresponds to an environment; instead, look for PSPs (walk the PSP chain, as it were), and look at offset 2Ch. Take the first environment you find. How do you know you have a PSP? Don't look for the opcode bytes for INT 20h (CDh 20h) like some programs do: instead, look for MCBs whose owner is one greater than the MCB itself. See the routine mstenvp2() in MSTENVP.C below.

Another technique, designed especially to accommodate command interpreters loaded with the /p option, also walks the MCB chain. This time you look for the *last* environment that is at a *higher* address than its corresponding PSP. (That's a mouthful.) Why? You look for the last environment because of the /p option, and you look for an environment higher than its PSP because the command interpreter builds an environment for itself: therefore the environment is at a higher address than the program. See mstenvp3() in MSTENVP.C below.

Finally, another technique involves walking back along the PSP chain until one finds a PSP which is its own parent. However, this technique (which first appeared in Barry Simon, "Providing Program Access to the Real DOS Environment," *PC Magazine*, 28 November 1989, pp. 309-314) is designed to find what is called the "active" environment, *not* the master environment. This method finds only the currently active shell: if you are within a program spawned from the ! statement in a dBASE or FoxPro program, for example, you won't find the master environment using this technique.

Three techniques for finding the master environment appear in the routine below. The function walk() in MSTENVP.C is a variation on the MEM.C program from chapter 3. Here, though, walk() expects a pointer to a function. For each MCB it finds, walk() will call this function, passing it an MCB pointer. The function should return TRUE, to indicate that walk() should keep going, or FALSE, to indicate that walk() should stop.

Thus, we can plug different functions into walk(), making MSTENVP.C a test bed for trying out different methods of finding the master environment via the MCB chain:

```
/*
MSTENVP.C
test bed for different methods of finding the master environment
*/

#include <stdlib.h>
#include <dos.h>

typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef unsigned long ULONG;
typedef void far *FP;

#ifndef MK_FP
#define MK_FP(seg,ofs)  ((FP)(((ULONG)(seg) << 16) | (ofs)))
#endif

#ifdef __TURBOC__
#define ASM asm
#else
#define ASM _asm
#endif
```

```
#pragma pack(1)

typedef struct {
    BYTE type;          /* 'M'=in chain; 'Z'=at end */
    WORD owner;         /* PSP of the owner */
    WORD size;          /* in 16-byte paragraphs */
    BYTE unused[3];
    BYTE dos4[8];
    } MCB;

MCB far *get_mcb(void)
{
    ASM mov ah, 52h
    ASM int 21h
    ASM mov dx, es:[bx-2]
    ASM xor ax, ax
    /* in both Microsoft C and Turbo C, far* returned in DX:AX */
}

#define MCB_FM_SEG(seg)     ((seg) - 1)
#define IS_PSP(mcb)         (FP_SEG(mcb) + 1 == (mcb)->owner)
#define ENV_FM_PSP(psp_seg) (*((WORD far *) MK_FP(psp_seg, 0x2c)))
#define PARENT(psp_seg)     (*((WORD far *) MK_FP(psp_seg, 0x16)))

char far *env(MCB far *mcb)
{
    char far *e;
    unsigned env_mcb;
    unsigned env_owner;

    if (! IS_PSP(mcb))
        return (char far *) 0;

    e = MK_FP(ENV_FM_PSP(mcb->owner), 0);
    env_mcb = MCB_FM_SEG(FP_SEG(e));
    env_owner = ((MCB far *) MK_FP(env_mcb, 0))->owner;
    return (env_owner == mcb->owner) ? e : (char far *) 0;
}

typedef enum { FALSE, TRUE } BOOL;
typedef BOOL (*WALKFUNC)(MCB far *mcb);

/*
    General-purpose MCB walker.
    The second parameter to walk() is a function that expects an
    MCB pointer, and that returns TRUE to indicate that walk()
    should keep going, and FALSE to indicate that walk() should
```

```
    stop.
*/
BOOL walk(MCB far *mcb, WALKFUNC walker)
{
    for (;;)
        switch (mcb->type)
        {
            case 'M':
                if (! walker(mcb))
                    return FALSE;
                mcb = MK_FP(FP_SEG(mcb) + mcb->size + 1, 0);
                break;
            case 'Z':
                return walker(mcb);
                break;
            default:
                return FALSE;   /* error in MCB chain! */
        }
}

/*********************************************************************/

/* using the GETVECT 2Eh technique (ENVPKG.ASM) */
void far *mstenvp1(void)
{
    ASM mov ax, 352eh       /* get INT 2Eh vector */
    ASM int 21h
    ASM mov dx, es:[002Ch]  /* environment segment */
    ASM xor ax, ax          /* return far ptr in DX:AX */
}

/*********************************************************************/

/* walk MCB chain, looking for very first environment */
void far *env2 = (void far *) 0;

BOOL walk2(MCB far *mcb)
{
    if (env2 = env(mcb))
    {
        unsigned psp = FP_SEG(mcb)+1;
        return (PARENT(psp) == psp) ? FALSE /*found it!*/ : TRUE;
    }
    else
        return TRUE;    /* keep going */
}
```

```
void far *mstenvp2(void)
{
    walk(get_mcb(), walk2);
    return env2;
}

/*********************************************************************/

/* walk MCB chain, looking for very LAST env addr > PSP addr */
void far *env3 = (void far *) 0;

#define NORMALIZE(fp)   (FP_SEG(fp) + (FP_OFF(fp) >> 4))

BOOL walk3(MCB far *mcb)
{
    void far *fp;
    /* if env seg at greater address than PSP, then
       candidate for master env -- we'll take last */
    if (fp = env(mcb))
        if (NORMALIZE(fp) > (FP_SEG(mcb)+1))
            env3 = fp;
    return TRUE;
}

void far *mstenvp3(void)
{
    walk(get_mcb(), walk3);
    return env3;
}

/*********************************************************************/

main()
{
    printf("GETVECT 2Eh method; mstenvp1   = %Fp\n", mstenvp1());
    printf("WALK MCB method; mstenvp2      = %Fp\n", mstenvp2());
    printf("WALK MCB/LAST method; mstenvp3 = %Fp\n", mstenvp3());
}
```

How does this program behave under various conditions?:

```
C:\UNDOC\KYLE>mstenvp
GETVECT 2Eh method; mstenvp1   = 0BC1:0000
WALK MCB method; mstenvp2      = 0BC1:0000
WALK MCB/LAST method; mstenvp3 = 0BC1:0000
```

So far so good: all three methods agree on where the master environment is. Now let's load a new permanent command interpreter with the poorly documented /p flag and run MSTENVP again:

```
C:\UNDOC\KYLE>mstenvp
GETVECT 2Eh method; mstenvp1    = 0CBD:0000
WALK MCB method; mstenvp2       = 0BC1:0000
WALK MCB/LAST method; mstenvp3 = 0CBD:0000
```

Here, mstenvp2() was in error: it stuck with the old abandoned environment segment in 0BC1:0000, instead of upgrading as the other two subroutines did. The /p flag creates a new primary command interpreter, not a secondary command interpreter. Strike one for mstenvp2().

Now, after loading a debugger that hooks INT 2Eh:

```
C:\UNDOC\KYLE>mstenvp
GETVECT 2Eh method; mstenvp1    = 7726:0000
WALK MCB method; mstenvp2       = 0BC1:0000
WALK MCB/LAST method; mstenvp3 = 0CBD:0000
```

Now, mstenvp1() is wrong, too: segment 7726 points into the debugger, not the primary command interpreter! So far, mstenvp3() is looking pretty good.

Now let's run MSTENVP from within a secondary command interpreter. For example:

```
C:\UNDOC\KYLE>command /c mstenvp
GETVECT 2Eh method; mstenvp1    = 0F57:0000
WALK MCB method; mstenvp2       = 0F57:0000
WALK MCB/LAST method; mstenvp3 = 129C:0000
```

Here, mstenvp3(), which until now looked pretty robust, failed miserably: it dutifully picked up the *last* MCB controlling an environment whose address was higher than its corresponding PSP: not surprisingly, that MCB belonged to the secondary command processor, not to the primary one. Oh well, the algorithm *sounded* good.

Finally, let's reboot the machine with DOS 4.0 (we've been running DOS 3.3), load a TSR in CONFIG.SYS with the INSTALL= command, and try MSTENVP again:

```
C:\UNDOC\KYLE>mstenvp
GETVECT 2Eh method; mstenvp1   = 13F9:0000
WALK MCB method; mstenvp2      = 0E4A:0000
WALK MCB/LAST method; mstenvp3 = 1349:0000
```

In this situation, mstenvp2() was wrong again: segment 0E4A points into the environment of the TSR loaded with the INSTALL= command! Thus, any program that uses the "walk MCB" method of finding the master environment will fail whenever a user of DOS 4 or higher uses the handy INSTALL= command. For example, if a .BAT file containing a SET statement is compiled with BAT2EXEC, it fails unexpectedly under this situation producing an "Out of environment space" message.

Could some enhancement be made to mstenvp2() to detect this situation? Yes: take, not the first environment you find, but the first environment belonging to a PSP that is its own parent (another mouthful!):

```
/* walk MCB chain, looking for very first environment belonging
   to PSP which is its own parent */
void far *env4 = (void far *) 0;

#define PARENT(psp_seg)     (*((WORD far *) MK_FP(psp_seg, 0x16)))

BOOL walk4(MCB far *mcb)
{
    if (env4 = env(mcb))
    {
        unsigned psp = FP_SEG(mcb) + 1;
        return (PARENT(psp) == psp) ? FALSE /*found it!*/ : TRUE;
    }
    else
        return TRUE;    /* keep going */
}

void far *mstenvp4(void)
{
    walk(get_mcb(), walk4);
    return env4;
}
```

This works beautifully in the DOS 4.x and higher INSTALL= situation, but it still leaves the problem with the /p option that caused mstenvp2() to fail.

Thus, although all these techniques work pretty well, none presents a 100-percent fool-proof method for finding the master environment. Of course, the problems discussed here are the result of some admittedly offbeat special cases, but anticipating such cases is what distinguishes the professional programmer from the amateur. So, what's a programmer to do? One technique, of course, is to perform the mstenvp() function in two different ways, compare the results, and bail out if they don't match.

It's some small comfort that Microsoft itself messed up the finding of the master environment in its own APPEND utility: the APPEND /E switch, when executed from a secondary command interpreter, does *not* affect the master environment and, in fact, can cause mysterious crashes.

Searching the Environment Just locating the environment block, of course, isn't enough to let you recover data from it. The next step is to know the format in which information is stored there, and to make use of that knowledge to find and retrieve the item you want.

The internal storage format used is, essentially, pure ASCII text. Each variable consists of a single ASCIIZ string. The first part of this string is the variable's name, and the rest is its data. The name and data are separated by an equal sign.

Each variable immediately follows the previous one, and the end of the list of variables is indicated by two consecutive all-zero bytes.

Since version 3.1, the EXEC function has added another item of information to the environment: the full path specification for the process that owns this copy of the environment. This information immediately follows the double-zero byte pair and begins with a binary (not ASCII) 01h word that indicates the number of items that follow. The drive letter, in ASCII, appears as the next byte, and the pathspec in ASCII continues until an all-zero end-of-string byte is reached.

Once you have a far pointer to the first byte of the environment area and know the internal storage format, it's simple to develop code to locate an environment variable by name.

In fact, current C compilers include library functions getenv() and setenv() to do just that, but they are limited to accessing only the current working copy of the environment. For maximum flexibility, you need to be able to access any copy you desire, because if you happen to be "shelled out" of a program, changes made to the current copy will vanish without trace when you return to the parent program. Changes made to the master copy will remain but will have no effect until you return to the primary shell.

Several methods could be used to provide totally flexible access to the environment; the one we'll explore is designed for easy expansion to other needs. Its foundation is the small assembly-language function shown in NXTEVAR.ASM, which follows:

```
;       NXTEVAR.ASM - Jim Kyle - July 1990

.model small,c

.code

nxtevar proc    uses di, vptr:far ptr byte

        public nxtevar
; char far * nxtevar( char far * vptr );
        les     di, vptr
        mov     cx, 8000h
        xor     ax, ax          ; search for 0 and...
        mov     dx, ax          ; ...initialize return DX:AX to 0:0
repne   scasb                   ; search ES:DI for char 0 in AL
        inc     cx              ; CX = 8000h if only one 0 found
        js      nev
        mov     dx, es
        mov     ax, di

nev:    ret
nxtevar endp
        end
```

NXTEVAR.ASM can be assembled either with MASM 5.1 or with Turbo Assembler in its MASM51 mode of operation. As presented, it's for use with small memory model C programs only, but the language interface can be changed easily by editing the ".model" line.

This function, when presented with a far pointer to any ASCIIZ string (not just one in the environment area), returns a far pointer to the byte that follows the end-of-string marker byte. In the environment's structure, that pointer is either to the first byte of the next string or to a byte that is all zeroes. In the latter event, the end of the environment's variable area has been reached, so nxtevar() returns the NULL pointer.

Before that happens, the previous call to nxtevar() will have returned the far pointer to the all-zeroes byte itself. If each pointer is retained in an array, the final

one can be used to recover the process' full path name by adding to its value, verifying that the next two bytes are then 01h and 00h, and finally copying the path information from the remaining bytes. Because the path data, like each variable, is an ASCIIZ string, the normal C string functions will deal with it properly.

This function is flexible because you can pass the function a pointer to any environment, including the current environment (obtained via the curenvp() function of ENVPKG) or the master environment (obtained by using mstenvp() instead).

To use NXTEVAR.ASM, you must assemble it into an OBJ file following the same procedures set forth earlier in this chapter for ENVPKG.ASM.

To illustrate how NXTEVAR.ASM is used, here's a little C program that reports the location and contents of each string in the current environment:

```
/*********************************************************
 *  NEV.C - Next Environment Variable                    *
 *  Jim Kyle, July 1, 1990                               *
 *********************************************************/
#include <stdio.h>
#include <stdlib.h>

char far * nxtevar( char far * vptr );
char far * curenvp( void );

void main (void)
{ char far * myenv;

  myenv = curenvp();
  while ( myenv )
    { printf("Env Var at %Fp: %Fs\n", myenv, myenv );
      myenv = nxtevar( myenv );
    }
  exit(0);
}
```

To create NEV.EXE from the C program, you can use either QuickC or Turbo C; both the ENVPKG.OBJ and NXTEVAR.OBJ modules must be used this time, because NEV.C calls curenvp() to locate the current environment. NEV.C tells you the address and contents of each variable string in the current environment, in the sequence in which they occur in the environment block. More often, you'll want to locate a specific environment variable by name, and you may want to locate it in the master environment rather than in the current copy. You could use

the library function getenv() to return a pointer to the named variable in the current environment copy, but not to access the master.

To both search by name and use the master environment, you can modify NEV.C into a far more useful program, FMEV.C, as follows:

```
/*********************************************************
 *  FMEV.C - Find Master Environment Variable            *
 *  Jim Kyle, July 7, 1990                               *
 *********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char far * nxtevar( char far * vptr );
char far * mstenvp( void );

void main ( int argc, char * argv[] )
{ char far * menv;
  char vname[128], *vdata, tgt[64];
  int tlen;

  menv = mstenvp();
  if (argc < 2)
    { printf("Var to find: ");
      gets( tgt );
    }
  else
    strcpy( tgt, argv[1] );
  tlen = strlen( tgt );

  while ( menv )
    { sprintf(vname, "%Fs", menv );
      if ( vname[tlen] == '=' )
        { vdata = &vname[tlen+1];
          vname[tlen] = '\0';
          if ( stricmp( tgt, vname ) == 0 )
            break;
        }
      menv = nxtevar( menv );
    }

  if ( menv )
    { printf("Found %s at %Fp:\n%s\n", vname, menv, vdata );
      exit(0);
    }
  else
```

```
    { printf("%s not found.\n", tgt );
      exit(1);
    }
}
```

In FMEV, the declaration of main() has been changed to permit the desired variable's name to be entered as a command-line argument, and sprintf() sidesteps the difficulties usually encountered when mixing far pointers and the small memory model.

Using sprintf() with the "%Fs" format specifier forces the library routine to do all necessary conversion to copy each string, in turn, from the environment to the local work area vname. Although some compilers include special library functions to do such mixed-pointer copying tasks, the sprintf() solution seems much simpler to comprehend.

The check for an '=' character in the position immediately following the last byte of the target string speeds the search by eliminating byte-by-byte comparisons unless the name and the target strings are the same length. In most environment areas, the savings in time is not significant, but this illustrates a programming technique that can speed up searches of larger areas, or for longer target strings.

FMEV, like the other C sample programs in this chapter, has been tested with both QuickC and Turbo C and requires either a PRJ file or a program list for use in the integrated development environment. The required OBJ files are NXTEVAR.OBJ and ENVPKG.OBJ.

How and Why COMMAND.COM Reloads Itself

One of the least understood parts of COMMAND.COM's internal operations is the area that controls the reloading of the transient portion upon return from an external command. If something goes wrong in this process, the error conditions range from confusing to downright misleading. And a point that's usually unclear is why the reloading happens sometimes, but not always.

Whenever an external command is loaded, the RAM occupied by the transient portion of COMMAND.COM is made available to that external command for use if needed, and upon return from an external command, the resident portion of COMMAND.COM does a checksum of the transient area to detect any changes in it. Although undocumented DOS function INT 2Fh Function 121Ch performs checksums, COMMAND.COM does not use this function. Instead, a

routine in the resident area accumulates the 16-bit sum in DX of all bytes in the transient area.

If any change has occurred, the entire transient area is reloaded from disk, using the COMSPEC environment variable to locate COMMAND.COM. Only the upper part of the file is loaded; the exact offset into the file at which the reload is to begin is "hard-wired" into the code that performs the reloading, and this offset varies from one DOS version to the next.

This checksum is first calculated for the transient portion immediately after the move to high RAM, and that result is stored in the resident portion. Each time the checksum is run after that, it is compared to the original value, and any mismatch causes the transient area to be reloaded. Thus, it's essential that the copy of COMMAND.COM that is pointed to by the COMSPEC variable be identical, byte for byte, to the copy that is used at boot time.

After the transient area is reloaded from disk, the checksum is run again to verify that the reload was in fact successful. Any mismatch at this time triggers an error message, "Unable to load COMMAND.COM, system halted," and the computer goes into a dynamic halt (JMP $) condition. This message usually indicates that the path set by COMSPEC is not valid, but the message can also be triggered by differences between the copy of COMMAND.COM reached via COMSPEC and the copy from which the checksum was calculated at boot time.

Such differences are most often caused by having mixed versions of COMMAND.COM on the system (that is, version 3.2 on the hard drive, but version 3.3 on the floppy from which the system was booted up), These problems may also be caused by patches applied to the disk copy (if a patch causes this problem, the reboot needed to use the system will clear things up, by causing a fresh copy of the checksum, which does include the patch effects, to be calculated).

Another possible cause of the reload error, not even hinted at by the message itself, occurs in network situations, when the network software redefines COMSPEC to point to the file server copy of COMMAND.COM (which may differ from the copy at any given workstation). The cure for errors resulting from this cause is to remove the redefinition of COMSPEC from the network software, if possible; if not, each workstation must run a batch file to perform the network login, and that batch file must fix up COMSPEC to point back to the workstation's own copy.

INT 2Eh, the Back Door

Interrupt 2Eh provides a "back door" into COMMAND.COM, which is used in the command interpreter itself for execution of batch files and which is used by the FOR and CALL internal commands. It's important to note that this interrupt is serviced by code in the resident portion of COMMAND.COM itself, rather than by DOS. If an alternative command interpreter such as 4DOS is installed in place of COMMAND.COM, the description provided in this section may be inapplicable.

Although INT 2Eh can be dangerous if you use it without knowing its limitations, it does permit access to the primary copy of the command interpreter shell (the one dispatched by CONFIG.SYS during system bootstrap).

The Function The purpose of INT 2Eh is to provide COMMAND.COM a method for accessing its own command parsing and dispatching routines. Originally, this facility supported only batch file processing and the FOR internal command. When the CALL internal command was added at DOS version 3.3, the INT 2Eh service routines were expanded to handle that also.

Because only COMMAND.COM supports this interrupt (although the author of 4DOS has indicated he plans to provide fully documented support in a future release, which may be available by the time this volume is published), it's only prudent to verify that the interrupt service routine exists before attempting to use it.

The following assembly language program, HAVE2E.ASM, provides a C-callable routine that returns TRUE if the first byte of the service routine for INT 2Eh is not the IRET that DOS points the vector to by default, and FALSE if it is equal to the IRET opcode.

```
.model small,c
.code
have2e  proc    ; returns 1 if ISR exists, else 0
        public  have2e
; int have2e( void ); /* prototype */

        mov     ax,352Eh
        int     21h
        mov     al,es:[bx]
        xor     al,0CFh         ; opcode for IRET
        jz      h1
        mov     ax,1
h1:     cbw
```

```
        ret

have2e  endp
        end
```

This function can be assembled using either MASM 5.1 or TASM; as presented, it is for the small memory model only, but this can be changed by modifying the ".model" line.

After assembly, copy the prototype line into your C programs (removing the semicolon that makes it a comment to the assembler) and include HAVE2E.OBJ with your C programs when linking. This is the same procedure described earlier in this chapter for use of the environment support modules.

Its Use To use INT 2Eh successfully, you must follow several rather strict guidelines. Because this capability has never been officially sanctioned, it omits many of the error-trapping abilities of more normal functions.

The most important restriction on use of this interrupt is that *no* registers, not even the stack segment and stack pointer, are preserved. Immediately at entry to the service routine, the return address offset and segment are popped into dedicated storage areas in COMMAND.COM's own data segment. Upon completion of the task invoked by using INT 2Eh, control returns to the caller by means of a far jump via the saved segment and offset values.

This has two major implications. The most obvious is that when you call INT 2Eh, you must save all essential registers in locations that you will be able to access upon return (when only CS and IP will be valid). Not so obvious is the fact that INT 2Eh, by its very nature, is not reentrant. That is, if it is called a second time before return from a prior invocation, the second call will destroy the return address for the first, leading to system lockup.

It is possible to overcome the lack of reentrancy. The trick is to copy all critical data areas from the COMMAND.COM data segment to dedicated memory space obtained specifically for the purpose, then invoke INT 2Eh, and upon return restore the data areas from the saved copy. This is essentially how the internal command CALL works, so that it can operate successfully from within a batch file.

The following program, DO2E.ASM, contains the function do2e(), which sets everything up properly for invoking the interrupt and regaining control properly upon return.

```
.model small,c
.code

do2e    proc    uses ds si di, cmdstr:ptr byte
        public  do2e
; void do2e( char * cmdstr ); /* prototype */

        mov     si,cmdstr       ; small model
;       lds     si,cmdstr       ; large model
        mov     cs:svss,ss
        mov     cs:svsp,sp
        cld
        int     2Eh
        cli
        mov     ss,cs:svss
        mov     sp,cs:svsp
        sti
        cld
        ret

        even            ; for best 286+ usage
svss    dw      0
svsp    dw      0

do2e    endp

        end
```

This routine should not be used unless have2e() returns TRUE to indicate that the interrupt is in fact present in your system. The command string passed to the do2e() function must be in a special format (the same format in which COMMAND.COM's input buffer is left after keyboard input): the first byte of the string must contain a binary count of the string's length (excluding the count byte and the terminating carriage-return character) and the string must be terminated by an 0Dh CR character.

Although you can build your own routines using only the do2e() function in DO2E.OBJ, it's easier if you work through an intermediate-level support module such as the following C function:

```
/*********************************************************
 *   Send2E.C - support for INT 2Eh                      *
 *   Jim Kyle, July 1990                                 *
 *********************************************************/
```

```
#include <stdio.h>
#include <string.h>

int have2e( void );                /* prototype */
void do2e( char * cmdstr );        /* prototype */

int Send2E( char * command )
{ char temp[130];
  int retval;

  if( retval = have2e() )
    { sprintf( temp, "%c%s\r", strlen( command ), command );
      do2e( temp );
    }
  return retval;
}
```

This snippet of code can be separately compiled, with the resulting OBJ file linked in your actual program, or it can be included directly in any program that uses it. Either way, it takes just the command line itself, as you would type it in at the prompt, and adds the character count and terminating CR. It then passes the edited string on to INT 2Eh using the assembly language module. These actions happen only if the interrupt is present; if not, they are skipped. Finally, Send2E() returns TRUE if the command was passed to INT2E and FALSE if it was not.

The following program runs Send2E() in a loop, making a little command interpreter:

```
/*
TEST2E.C
Turbo C++ 2.0:
    tcc test2e.c send2e.c do2e.asm have2e.asm
Microsoft C 6.0:
    cl -qc test2e.c -MAmx send2e.c do2e.asm have2e.asm
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

main()
{
    char buf[80];
    for (;;)
    {
```

```
        fputs("$ ", stdout);
        gets(buf);
        if (strcmp(buf, "bye") == 0 || strcmp(buf, "BYE") == 0)
            break;
        Send2E(buf);
    }
    puts("Bye");
}
```

We already saw this program in chapter 5 on TSRs, where it was used with Ray Michels' TSR skeleton to create a memory-resident command interpreter, TSR2E.

It is worth noting that this program—and INT 2Eh in general—is fully compatible with the installable-command interface discussed earlier in this chapter. For example, if you run our INSTCMD program, you can issue its new internal command FULLNAME via an INT 2Eh program such as TEST2E or TSR2E.

An interesting thing happens if a program uses Send2E() to issue a SET command: the master environment is updated instead of the local copy of the environment belonging to the program. Using INT 2Eh is thus one technique that can be used for updating the master environment; other, safer, techniques were detailed earlier in this chapter, in the section "Using the Environment."

Notice that the issue of reentrancy is not addressed in Send2E(). That's because the issue is much more complex than just saving the data areas and restoring them. Although that's necessary, other considerations must also be addressed to make INT 2Eh calls robust enough for dependable use.

The first question that arises is how much data to save from the DOS area. In the only significant article published on this subject, Daniel E. Greenberg ("Reentering the DOS Shell," *Programmer's Journal*, May/June 1990) suggested that 120 bytes would include all necessary information for DOS versions 2 through 4.01. However, not all of that 120 bytes needed to be restored, and the locations that required restoration varied from one DOS version to the next.

Another question, equally important, concerns the methods required to handle CTRL-BREAK interruptions, or critical error conditions. In both cases, the normal COMMAND.COM response is inadequate to provide full safety.

Greenberg presented a major program written in assembly language to accompany his article, and a study of it is recommended if you want to use INT 2Eh as a tool for serious programming. However, because future revisions of DOS will undoubtedly cause changes to this capability at least as significant as

those that have accompanied each revision in the past, your safest course is to avoid the use of INT 2Eh entirely, aside from its function in guiding you through some of the undocumented internal workings of COMMAND. COM.

Alternatives to COMMAND.COM

Several alternatives to COMMAND.COM now exist, and this section looks at a sampling of them. Some provide complete replacement of the command interpreter, while others retain the existing command interpreter but hide its command-line interface from the user.

4DOS.COM

This command interpreter, from J. P. Software, totally replaces COMMAND. COM. Originally distributed only as shareware, it is now also available through some retailers. As this is written, it's at version 3.01; another major upgrade, is expected to be released before this volume is published.

Other products that replace COMMAND.COM include Command-Plus, PolyShell, and FlexShell. I chose to describe 4DOS because it typifies the group, and also because I use it daily and am more familiar with it than with the others (all of which, however, offer improved capabilities when compared to COMMAND.COM).

A Total Replacement, Plus More Like all its competitors, 4DOS provides near-total compatibility with COMMAND.COM, even going so far as to duplicate strange actions that most users consider to be "bugs" in order to maximize the number of applications that can run without change.

Where COMMAND.COM has only 37 internal commands at most, however, 4DOS provides more than twice that many, thereby making many utilities used with COMMAND.COM obsolete.

For example, this command interpreter includes a built-in environment editor. It also includes built-in ALIAS capabilities and automatic command-line HISTORY recall (to obtain these functions with COMMAND.COM, many power users run DOSEDIT or CED). For building batch-file menuing systems, internal commands are able to draw boxes on the screen and obtain input from the keyboard that then modifies batch-file execution decisions.

One of the most useful enhancements is the ability to do batch-file processing entirely in RAM. COMMAND.COM must reload a batch file from disk for each

line of input, making it necessary to keep the file available throughout its processing. Reading the file into RAM makes it possible to remove a floppy-disk-based file from the system physically, yet continue its execution.

Unlike COMMAND.COM, 4DOS does not split itself apart during installation. The system actually consists of two separate files, one of which forms the resident portion (and holds the data) while the other serves as the transient portion and is swapped out of RAM, when any external command is dispatched.

The swapping techniques used are one area in which 4DOS has greatly improved on the "standard" command interpreter. Rather than reload from the same copy that was used to load the program initially, 4DOS actually swaps out the entire transient area of RAM including any data storage that's not necessary for swapping it back in. This makes it possible for the program to shrink itself to only 256 bytes in normal DOS RAM, when run on a 286 or higher system that has the ability to load the resident portion in "high memory."

Command-line switches supplied on the SHELL= line in CONFIG.SYS control just how 4DOS does its swapping; you have the choice of swapping to the High Memory Area, to Expanded Memory, to disk, or not swapping at all. You can also specify the size of your environment area and of the buffer in which command line history will be saved.

Not the least of the enhancements, incidentally, is a full-featured hypertext-like HELP system that gives you full details of how to use each of the internal commands, from the command line. With all the added power, this feature is needed often.

No Undocumented Features One of the most amazing things about 4DOS is that it is implemented entirely with documented features of DOS and works across all versions of DOS from 2.0 on. It makes no use of the undocumented features and hooks on which COMMAND.COM depends for success.

Planned for inclusion soon, possibly by the time you read this, is full implementation of INT 2Eh as a "back door" into the command interpreter, together with documentation of the capability. In version 3.01, however, INT 2Eh is vectored only to an IRET command. It is, however, part of the 4DOS code segment, so that the interrupt vector can be used to locate the primary shell, just as with the "standard" command interpreter.

Menuing Systems

Rather than totally replacing COMMAND.COM with some other command interpreter, many developers have created systems that allow end users to choose what they want to do from one or more menus displayed on the screen. These menu-based systems are often called *shells*, although that term, strictly speaking, should be reserved for the entire command interpreter.

Menuing systems span a range from something based on just a few batch files to a baroque and somewhat Rube-Goldberg-like arrangement involving a batch file, a COM file, an EXE file, a number of MENU files, and near-total lack of documentation. In this section we briefly examine both extremes.

Batch-File Menu Systems The simplest method of insulating an unsophisticated end user from the perils of the command-line prompt is to create a full-screen menu that presents the user all the options needed and permits the choice of any one of them. And the simplest way to put together such a menu is by means of a set of batch files, working together.

Thousands of such simple menu shells are in use, and entire books have been written about this single subject. Although it's possible to create menuing systems that are much more complicated, there's really very little reason to do so.

That did not, however, keep exponents of the "bigger is better" school of program design from trying, as you shall see.

The DOSSHELL Approach As part of MS-DOS 4.0, a menuing environment built around a command called DOSSHELL was introduced. This capability involved a strange division of effort between COMMAND.COM, DOSSHELL.BAT, SHELLB.COM, and SHELLC.EXE, to accomplish what many third-party programs do, using only batch files.

In order to launch the DOSSHELL feature, the command "DOSSHELL" is typed in either as the final command of the AUTOEXEC.BAT file or on the COMMAND.COM command prompt. Either way, COMMAND.COM gets the command, and dispatches it as a possible external command.

That action finds the file DOSSHELL.BAT, which was created during the installation process for DOS 4.0; you won't find such a file on your distribution disks, and if you bypass the official INSTALL process you will not be able to use DOSSHELL. If DOSSHELL.BAT is found, it then begins execution by changing your current working directory to the one in which INSTALL placed your DOS utility files and then invoking SHELLB with the argument "DOSSHELL".

This command is interpreted by COMMAND.COM; it also turns out to be an external command that invokes the SHELLB.COM program and passes it the name of the executing batch file. SHELLB then installs its own handler for Multiplex Interrupt 2Fh, Functions 19h and AEh. The first of these functions provides communication between COMMAND.COM, SHELLB's resident portion, and SHELLC.EXE (which we have not yet met). Function AEh (Installable Command), discussed earlier in this chapter, lets COMMAND.COM treat portions of SHELLC.EXE's code as if they were internal commands within COMMAND.COM.

Once SHELLB has installed the interrupt handler and has gone resident, it returns control to DOSSHELL.BAT. If it did not return an error code of 255, DOSSHELL.BAT then disables CTRL-BREAK checking by DOS and invokes SHELLC with a long string of option switches that specify such things as the video driver to use, the menu to display initially, and other details.

Four undocumented groups of options for use on this line have recently been revealed. No more than one option from each group should be specified; use of these options changes the default conditions for DOSSHELL actions:

```
/CONFIRMDELETEON
/CONFIRMDELETEOFF

/CONFIRMREPLACEON
/CONFIRMREPLACEOFF

/ALLOWSELECTON
/ALLOWSELECTOFF

/SORTBYNAME
/SORTBYEXT
/SORTBYSIZE
/SORTBYDATE
/SORTBYDISK
```

SHELLC.EXE is the actual menu display module, and it normally remains resident so long as DOSSHELL is in use. It displays the menu and dispatches selected menu items back to COMMAND.COM as commands for execution, much as a batch file would do. It is, however, much hungrier in its appetite for memory space, and much more difficult to troubleshoot or to modify. Its only real advan-

tage is that it lets you add explanatory "help" paragraphs to each displayed menu option that pop into view when that option is highlighted.

Small wonder, then, that this whole function is being totally revamped in the next version of DOS. Because it apparently will be only a one-version aberration, we won't go into additional detail about it.

Sample Program: Master Environment Editor

It's time to put everything together into a single sample program that illustrates this chapter's topics and that is also potentially useful. The program, ENVEDT.C, lets you edit the master environment no matter how many levels down you happen to be shelled. ENVEDT can be used with any command interpreter that vectors INT 2Eh to its own code space and that follows COM-file conventions (that is, CS and DS point to the same segment address).

ENVEDT first locates the master environment block and then locates the specific variable to be edited. If you fail to give it a variable name, it displays a brief summary of how it is to be used, followed by a list of all variable names currently contained in the master environment.

Most of the specific techniques that ENVEDT uses have already been explained earlier in this chapter; the value of this program is that it shows you how to put the pieces together. In addition to ENVEDT.C itself, the program requires the support routines in ENVPKG.ASM and NXTEVAR.ASM, plus a new module, EEA.ASM (Environment Editor assembly code).

ENVEDT.C has been tested both with Turbo C and with Microsoft QuickC, and EEA.ASM has been tested with Turbo Assembler and with MASM V5.1. The finished program has been tested with both COMMAND.COM and 4DOS.COM as the primary shells, and with DOS versions ranging from 3.2 to 4.01.

```
/*********************************************************
 *  ENVEDT.C - Editor for Master Environment Variables    *
 *  Jim Kyle, July 8, 1990                                *
 *                                                        *
 *      qcl envedt.c eea.obj envpkg.obj nxtevar.obj       *
 *   or tcc envedt.c eea.asm envpkg.asm nxtevar.asm       *
 *                                                        *
 *********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
```

```
#ifndef FP_SEG
#define FP_SEG(f) (*((unsigned *)&(f) + 1))
#endif

#ifndef FP_OFF
#define FP_OFF(f) (*((unsigned *)&(f)))
#endif

#ifndef MK_FP
#define MK_FP(s,o) ((void far *)\
     (((unsigned long)(s) << 16) | (unsigned)(o)))
#endif

extern char far * nxtevar( char far * vptr );
extern char far * mstenvp( void );
extern void max_xy( int *x, int *y );
extern int col( void );
extern int row( void );
extern void setrc( int r, int c );
extern int envsiz( char far * envptr);

char far * menv;                /* pointer to current var */
char far * rest;                /* pointer to next var    */
char far * lstbyt;              /* adr of last byte used  */
char vname[512], *txtptr;       /* working buffer and ptr */
int  nmlen,  insmode = 0,       /* name len, insert flag  */
     max_x,  max_y,             /* screen limits          */
     i,      c,                 /* scratchpads            */
     begrow, begcol,            /* cursor, start of text  */
     currow, curcol,            /*         current loc    */
     endrow,                    /*         end of text    */
     editing,                   /* loop control flag      */
     i_cur, i_max,              /* cur, max i in txtptr   */
     free_env;                  /* bytes free in env      */

void findvar( char * varnam )   /* find var, set txtptr   */
{ nmlen = strlen( varnam );
  txtptr = NULL;                /* present not-found flag */
  while ( *menv )
    { rest = nxtevar( menv );   /* "rest" always next one */
      sprintf( vname, "%Fs", menv );
      if( vname[nmlen] == '=')  /* possible match found   */
        { vname[nmlen] = '\0';
          if (stricmp( vname, varnam ) == 0)
            { txtptr = &vname[nmlen+1];
              vname[nmlen] = '=';
              return;           /* found it, get out now  */
```

```
            }
        }
      menv = rest;                /* try again with next    */
    }
}

void calccrsr( void )             /* calc currow, curcol    */
{ begrow = endrow - (i_max / max_x );
  if (( i_max % max_x ) == 0 )  /* correct for line wrap  */
    begrow++;
  begcol = 0;
  currow = begrow + (i_cur / max_x );
  curcol = begcol + (i_cur % max_x );
}

void show_var( void )             /* display var content    */
{ setrc( begrow, begcol );        /* set to start           */
  printf( txtptr );               /* show the string        */
  endrow = row();                 /* update end row if scrl */
  if( ! col() )                   /* adjust for line wrap   */
    endrow--;                     /*   if now in col 0      */
  calccrsr();                     /* establish cursor posn  */
}

void do_del( void )
{ for (i=i_cur; txtptr[i]; i++ )/* slide over one to left */
    txtptr[i] = txtptr[i+1];
  if ( i_max && i_cur >= --i_max )       /* decr length    */
    i_cur = (i_max - 1);          /* and adjust if needed   */
  free_env++;                     /* account for freed byte */
  setrc( begrow, begcol );        /* re-display the string  */
  printf( txtptr );
  endrow = row();                 /* hold ending point      */
  if( ! col() )                   /* adjust for line wrap   */
    endrow--;                     /*   if now in col 0      */
  putchar( ' ' );                 /* erase garbage char     */
  calccrsr();                     /* establish cursor posn  */
}

void dochar( void )
{ if ( free_env < 3 )             /* just beep if no space  */
    { putchar( 7 );
      return;
    }
  if ( insmode )                  /* open up a hole for new */
    for (i = ++i_max; i > i_cur; i--)
      txtptr[i] = txtptr[i-1];
```

```
  txtptr[i_cur++] = (char) c;   /* put char down          */
  if ( i_cur == i_max )         /* check for extending it */
    { txtptr[ ++i_max ] = '\0'; /* set new EOS            */
      free_env--;               /* then count down space  */
    }
  show_var();                   /* re-display the string  */
}

int edtxt( void )               /* read kbd, do editing   */
{ int retval;
  begrow = row();
  begcol = 0;
  i_max = strlen( txtptr );     /* set buffer index limit */
  i_cur = 0;                    /* and current index val  */
  show_var();                   /* display the string     */
  for ( editing=1; editing; )   /* main editing loop here */
    { setrc( 0, 70 );           /* status message loc     */
      printf("MODE: %s ", insmode ? "INS" : "REP" );
      setrc( currow, curcol );  /* keep cursor posn curr  */
      switch( c = getch() )
        { case 0:               /* function key or keypad */
            switch( getch() )
              { case 30:        /* Alt-A, re-display       */
                  show_var();   /* re-display the string  */
                  break;
                case 32:        /* Alt-D, delete variable */
                  printf("\nDELETE this variable (Y/N)? ");
                  if(( getch() & 89 ) == 89 ) /* 89 = 'Y' */
                    { vname[0] = '\0';
                      retval = 1;
                      editing = 0;
                    }
                  break;
                case 71:        /* home, goto first char  */
                  i_cur = 0;
                  calccrsr();   /* establish cursor posn  */
                  break;
                case 72:        /* up arrow               */
                  if ( (i_cur - max_x ) > 0 )
                    i_cur -= max_x;
                  calccrsr();   /* establish cursor posn  */
                  break;
                case 75:        /* left arrow             */
                  if ( i_cur > 0 )
                    i_cur--;
                  calccrsr();   /* establish cursor posn  */
                  break;
```

```
                case 77:        /* right arrow            */
                  if ( i_cur < i_max )
                    i_cur++;
                  calccrsr();   /* establish cursor posn  */
                  break;
                case 79:        /* end, goto last char    */
                  i_cur = ( i_max ? i_max - 1 : 0 );
                  calccrsr();   /* establish cursor posn  */
                  break;
                case 80:        /* down arrow             */
                  if ( (i_cur + max_x ) < i_max )
                    i_cur += max_x;
                  calccrsr();   /* establish cursor posn  */
                  break;
                case 82:        /* insert, toggle flag    */
                  insmode = !insmode;
                  break;
                case 83:        /* delete, remove 1 char  */
                  do_del();
                  break;
              }                 /* end of special codes   */
            break;
          case 8:               /* backspace del to left  */
            if (i_cur)
              { i_cur--;        /* back up one first      */
                do_del();       /* then do the delete     */
              }
            break;
          case 13:              /* Enter accepts changes  */
            retval = 1;
            editing = 0;
            break;
          case 27:              /* ESC quits without save */
            retval = 0;
            editing = 0;
            break;
          default:
            if (c >= ' ' && c < 127)
              dochar();         /* handle INS or REP      */
            else
              putchar( 7 );     /* beep on any other char */
        }
    }
  setrc( endrow, 0 );
  return (retval);
}
```

```
void putenvbak( void )          /* copies back to env     */
{ char * locptr;
  int save_size;

  save_size = FP_OFF( lstbyt ) - FP_OFF( rest ) + 1;
  locptr = (char *)malloc( save_size );

  for( i=0; i<save_size; i++ )  /* save trailing data     */
    locptr[i] = rest[i];
  for( i=0; vname[i]; i++ )     /* copy edited string     */
    *menv++ = vname[i];
  if( vname[0] )                /* if not deleting...     */
    *menv++ = '\0';             /* ...add EOS byte to var */
  for( i=0; i<save_size; i++)   /* copy in trailing data  */
    *menv++ = locptr[i];
  free( locptr );               /* release save area      */

  printf("\nENVIRONMENT UPDATED." );
}

void doedit( char * varnam )    /* find var, edit, save   */
{ printf("Editing '%s':\n", varnam );
  menv = mstenvp();             /* set starting point     */
  free_env = envsiz(menv) << 4; /* get the size in bytes  */
  findvar( varnam );            /* look for the variable  */
  for( lstbyt=menv; *lstbyt; )  /* menv set by findvar()  */
    lstbyt=nxtevar(lstbyt);     /* locate end of var area */
  if( lstbyt[1] == 1 && lstbyt[2] == 0 )
    { lstbyt += 3;              /* skip loadfile name     */
      while (*lstbyt)
        lstbyt++;
    }
  lstbyt++;
  free_env -= FP_OFF( lstbyt ); /* what's left is free    */
  if ( txtptr == NULL )         /* didn't find the name   */
    { free_env -= (nmlen+1);    /* take out free space    */
      if ( free_env < 5 )
        { puts("Not found, no room to add.");
          return;
        }
      printf( "Not found; add it (Y/N)? " );
      if(( getch() & 89 ) != 89 )       /* 89 = 'Y'       */
        return;
      for ( i=0; i<nmlen; i++ ) /* force to uppercase     */
        vname[i] = (char) toupper( varnam[i] );
      vname[nmlen] = '=';       /* add the equals sign    */
      vname[nmlen+1] = '\0';    /* make content empty     */
```

```
      txtptr = &vname[nmlen+1]; /* set text pointer to it */
      putchar( '\n' );          /* start on fresh line    */
      insmode = 1;              /* and in INS mode        */
    }
  printf("Free environment space = %d bytes.\n", free_env );
  if ( edtxt() )                /* do the editing now     */
    putenvbak();                /* copy to master env     */
  else
    printf("\nENVIRONMENT NOT CHANGED." );
  putchar( '\n' );
}

void showvars( void )           /* prints usage message   */
{
  puts(" USAGE: ENVEDT varname [[name2] ... ]");
  puts("where varname is the name of an env variable");
  puts("  and name2, etc., are optional added names.");
  puts("Current variable names are:" );
  menv = mstenvp();
  for( i=0; i<8; i++ )
    vname[i] = ' ';
  while ( *menv )               /* get and print names    */
    { sprintf(vname+8, "%Fs", menv );
      for( i=8; vname[i] != '='; i++ )
        /* all done by for()        */ ;
      vname[i] = '\0';
      puts( vname );
      menv = nxtevar( menv );
    }
   puts("Re-run with name(s) of variable(s) to be edited.");
}

void main ( int argc, char **argv )
{ int i;

  if (argc < 2)
    showvars();                 /* list all vars to CRT   */
  else
    { max_xy( &max_x, &max_y ); /* set up screen limits   */
      while ( --argc )          /* process all vars named */
        doedit( *++argv );
    }
}
```

If the only thing on the command line is the program name itself (argc < 2), the showvars() procedure is called. If one or more additional arguments were

present, the doedit() function is called for each of them in turn until all have been processed.

The showvars() procedure locates the master environment using mstenvp(), and cycles through it via nxtevar(), displaying the name of each variable. The SET internal command does almost the same thing, but if you are not working in the primary shell, SET deals with the local copy of the environment rather than with the master. ENVEDT always works on the master copy.

The doedit() function searches the master environment for a variable, again using mstenvp() to start each search at the front of the master environment area. Before searching, it sets the variable free_env to the total size in bytes of the master environment block, using envsiz(), so that the amount of free space can be calculated later. The function then calls findvar() to do the actual search.

The findvar() procedure sets a pointer, txtptr, either to the address of the variable's contents in global work buffer vname (if it is found) or to NULL. This tells doedit() whether the search was successful. Before checking the result, however, doedit() moves another pointer, lstbyt, past any remaining variables and past the primary shell's loadfile pathspec, if one is present (normally, none is). When this is done, the offset portion of lstbyt is the count of the total number of environment-block bytes in use; subtracting it from the total size of the block leaves in free_env, as the remainder, the number of bytes still free for use.

With free_env properly calculated, doedit() then checks the search result. If the name was not found, the procedure asks if you want to add it as a new variable (first verifying that there's room to do so and correcting the free_env value to account for the name itself). If you do, the name is copied into the global work buffer vname, the '=' character and a terminating '\0' are added, txtptr is set to the address of the '\0' character (the presently empty new contents), and the insmode flag is set to indicate INSERT mode operation. Otherwise, doedit() returns to main() to process the next variable in the input list.

If the variable was found, or if a new one is to be added, doedit() reports the number of bytes still free to use and calls edtxt() to do the actual editing. If edtxt() returns a nonzero value (indicating normal completion of the editing operation), putenvbak() is called to copy the work buffer vname back into the master environment block, sliding other data around as necessary to make things fit. If the value returned by edtxt() is zero, indicating an ESC-key bailout, the message "ENVIRONMENT NOT CHANGED" is displayed and doedit() returns.

Before we look at edtxt(), which contains the bulk of ENVEDT's complexity, let's see how findvar() searches for the name in the master block. The three pointers that address the master block (menv, rest, and lstbyt) are all declared as far pointers and are global in scope so that all procedures in ENVEDT can use them. The findvar() routine gets a near pointer (actually, one of those in the argv array) as its argument, and the menv pointer is initialized to the first byte of the master block before the first call to findvar() is made.

Each time that findvar() is entered, the pointer txtptr is set to NULL and the global variable nmlen is set to the length of findvar()'s argument. Then findvar() goes into a loop that continues until either the end of the variable list is reached (when the byte pointed to by menv will be zero) or the argument is found as a variable name.

Within this loop, pointer "rest" is first set to the address of the next variable, using nxtevar(). The current variable is then copied from the master block to global work buffer vname by means of the sprintf() library function and its "%Fs" format modifier. Next, the character at position "nmlen" in the vname buffer is checked; if it is '=', the variable's name is the right length to be a possible match; if not, no match is possible so no time is wasted trying to compare the strings.

If the length is right, the '=' is temporarily replaced by '\0' and the library function stricmp() is used to compare the name and the argument without regard to case. If they match, the '=' is put back, txtptr is set to the first byte after it, and findvar() returns successfully. If the lengths differ or the strings fail to match, "rest" is copied to "menv" and the process repeats for the next variable in the master block.

Upon successful return from findvar(), menv still points to the first byte of the variable being edited, the vname buffer contains an exact copy of the entire variable, including its name and the trailing '=' separator, and txtptr points to the first byte of its value, in vname. These facts are critical to the operation of edtxt(), which does the actual editing.

On entry to edtxt(), variables are set up to save the cursor position and show_var() is called to display the variable on the screen. The main loop is then entered, and control remains within this loop until you press Alt-d, Enter, or Esc. Any of these keys clears control variable editing; Enter and Alt-d set the return value to 1, and Esc sets the return value to 0. The cursor is then positioned just past the end of the variable on the screen, and control returns to doedit().

Each time that show_var() is called, it positions the cursor to the saved starting position, displays the entire contents of the variable (using txtptr), saves the ending cursor position, and calls the function calccrsr() to calculate any necessary adjustments to the saved starting points and to establish the current cursor position within the variable.

The Alt-A keystroke simply refreshes the display should anything cause it to become confused. The Alt-D entry tells ENVEDT that you want to delete the variable entirely rather than just changing it. When you have verified that you really mean it, the program zeroes the first byte in vname[], and forces the same actions as those that occur when Enter is pressed, so that no bytes move back to the master block but any variables following close up the gap.

The arrow keys move the cursor as would be expected, with the restriction that it cannot be moved out of the variable. That is, up-arrow has no effect if the cursor is on the top line of the variable, left-arrow does nothing if the cursor is on the first character, and so forth. The Home and End keys move the cursor to the first and last characters, respectively, of the variable. The code for all six of these keystrokes simply changes the value of i_cur as appropriate, then calls calccrsr() to do the heavy work.

Once edtxt() has completed and returns a nonzero value, putenvbak() handles the job of moving the edited variable back into the master block. It does this by first calculating the number of bytes that must be saved, using the values of "rest" and "lstbyt" that were set up earlier, and copying that material from the master block into a temporary block of memory obtained via malloc().

The full content of vname is then copied back to the master block starting at the address indicated by menv. Finally, the saved material is copied in following the vname data and the temporary block released by free(). The final action of putenvbak() is to display on screen the line "ENVIRONMENT UPDATED."

The module EEA contains functions that are used when you are interactively editing the environment variables:

```
;       EEA.ASM - support for EnvEdt.C

.model small,c

.code

max_xy  proc    x:word,y:word
        public  max_xy
```

```
; void max_xy( int *x, int *y );
        mov     ax,1130h        ; try EGA/VGA routines
        xor     dx,dx
        push    bp              ; save BP around INT 10h
        int     10h             ; in case we're running on
        pop     bp              ; an old BIOS that trashes BP
        or      dl,dl
        jnz     mxy2            ; nope, not EGA or VGA
        mov     dl,24           ; so set for 25 lines
mxy2:   xor     dh,dh
        inc     dx
        mov     bx,y
        mov     ds:[bx],dx      ; store maxy value
        mov     ah,0fh          ; use bios mode call
        push    bp
        int     10h
        pop     bp
        xchg    ah,al
        cbw
        mov     bx,x
        mov     ds:[bx],ax      ; store maxx value
        ret
max_xy  endp

col     proc
        public  col
; int col( void );
        mov     ah,3            ; Get Cursor Position
        xor     bx,bx
        push    bp
        int     10h
        pop     bp
        mov     al,dl           ; return x coordinate
        cbw
        ret
col     endp
row     proc
        public  row
; int row( void );
        mov     ah,3            ; Get Cursor Position
        xor     bx,bx
        push    bp
        int     10h
        pop     bp
        mov     al,dh           ; return y coordinate
        cbw
        ret
```

```
row     endp
setrc   proc    r:byte, c:byte

        public  setrc
; void setrc( int r, int c );
        xor     bx,bx
        mov     dl,c
        mov     dh,r
        mov     ah,2            ; Set Cursor Postion
        push    bp
        int     10h
        pop     bp
        ret
setrc   endp

        end
```

To some degree, these functions duplicate others found in the Turbo C and QuickC libraries, but the library functions differ greatly between the two compilers. The addition of these video routines simplified the rest of the program by providing identical operation regardless of the compiler you choose.

Conclusion

In this chapter, we have looked at a wide variety of topics: the basic structure of the input-evaluate-do loop found in any command interpreter; the DOS hooks for installing new internal commands; the skeletal structure of a DOS command interpreter (TSHELL); the division of COMMAND.COM into initialization, resident and transient portions; the DOS environment; the command interpreter backdoor (INT 2Eh); alternative interpreters like 4DOS; and editing the master environment.

That we have covered so many topics reflects the nature of command interpreters: they are, after all, *interpreters* of human input and produce (hopefully) human-readable output. While the basic operation is simply an input-evaluate-do for (;;) loop, there are enough different forms that input can take (from the keyboard, batch file, environment variable, or via the interpreter backdoor), and enough different locations for command code (internal, external, and installable), that we could have easily made this chapter even longer if we had wanted.

It is also important to remember that, in MS-DOS as in Unix, the command interpreter or shell is not part of the operating system itself. This is one reason why some of the techniques presented in this chapter will not work on every

DOS machine. You can take a good many of the programs in this book, walk up to any of the 30 million DOS machines in existence, and the program will run. Not necessarily so with the programs in this chapter, though: if a machine is running some interpreter other than COMMAND.COM (such as, heaven forbid, our own little TSHELL.COM), then techniques such as installing new internal commands, finding the master environment, or invoking INT 2Eh may or may not work.

This is because anyone is free to throw away COMMAND.COM and substitute something completely different. Now, it's true that users are also free to change parts of MS-DOS itself (by hooking INT 21h, for example), and it's also true that such changes are *more* widespread than using alternate command interpreters. But still, the command interpreter is *not* part of the operating system, and it's important to remember this.

In other words, there are no 100% guarantees here. But then again there can be none anywhere in an operating system as flexible as MS-DOS.

Chapter 7

The MS-DOS Debugger Interface

Tim Paterson

Some tools would be virtually impossible to write for the MS-DOS environment using only the documented interface to DOS. A debugger, profiler, or performance analyzer must be able to execute another program under its own control—but not by merely using the DOS EXEC function (INT 21h Function 4B00h) to fire up independent execution.

The most basic need is to load the "child" program, ready for execution, without actually executing it. This function, plus a few others described in earlier chapters, is enough to make a basic DOS debugger. The first part of this chapter will show you the details, using as an example a DOS debugger, similar to DEBUG, called Monitor.

If we stopped there, a whole world of programs would still be out of reach of our hypothetical debugger. The Microsoft Windows environment adds a new level of complexity to any tool that tries to observe the execution of a program. To make optimum use of limited memory, Windows is constantly loading, discarding, and moving code segments around in memory. The second part of this chapter will show you how we can give Monitor a direct link to Windows to keep it informed of all this activity.

Loading Without Executing

The ability to *load* a program for execution—without actually executing it—is an essential requirement for any debugger. A function to do this is present, although not documented, in all versions of MS-DOS since 2.0. In fact, the software base that uses this function is quite large, including Microsoft's DEBUG, SymDeb, and CodeView debuggers. Note, however, that the description here applies only to DOS versions 3.0 and later.

Revealing a Subfunction

The function for loading a program without executing it is actually just one of the subfunctions of the standard DOS EXEC function (INT 21h Function 4Bh). Published documentation describes two subfunctions:

- AL = 0 Load and execute program
- AL = 3 Load overlay

To this list we now add:

- AL = 1 Load program

For all EXEC subfunctions, additional arguments are passed in registers:

- DS:DX Pointer to filename, zero terminated
- ES:BX Pointer to parameter block

The parameter block pointed to by ES:BX is a little longer for the undocumented Load Program subfunction than it is for Load and Execute. Two additional far pointers have been tacked on to the end. Unlike the rest of the parameter block, which is used to pass values to DOS, these additional entries are used for values returned *from* DOS. The first of these two far pointers is the initial value of SS:SP; the second is the initial value for CS:IP:

```
ExecBlock   struc

Environment dw        ?
CommandTail dd        ?
FCB1        dd        ?
FCB2        dd        ?
InitStack   dd        ?     ; SS:SP Filled in by DOS
InitIp      dd        ?     ; CS:IP Filled in by DOS

ExecBlock   ends
```

The return conditions are exactly the same as for Subfunction 0 (Load and Execute) and are shown in the appendix.

The documentation for Subfunction 0 (Load and Execute) claims that all other registers are preserved. However, this is not the case. Both Subfunctions 0 and 1 return with DX and BX registers destroyed. (In DOS 2.x, *all* registers, including SS:SP, were destroyed.)

Upon successful return from this function, the current Program Segment Prefix (PSP) will be set to the newly loaded program. You may need to switch back and forth between the child's PSP and the parent's PSP, as we'll discuss shortly. In addition, the child's termination address is set to the same location in the parent that DOS just returned to—that is, the first instruction after the INT 21h used to invoke the Program Load function. We'll also see more about this important topic a little later.

Preparing the ExecBlock

Now it's time to jump in and take a closer look at what you need to really use this function. Start by making sure you've set up the call to DOS correctly, paying particular attention to the contents of the ExecBlock. This part, at least, is no different than the normal (documented) Load and Execute subfunction.

The first entry in the ExecBlock is the segment of the environment. DOS uses this to provide a copy of the environment to the child process. Normally, you will just want to pass on the same environment as the parent. DOS makes this easy by accepting the value of zero as a flag that indicates the parent's environment should be used. No code is required since the ExecBlock is statically initialized to zero.

The command tail and FCB pointers in ExecBlock are used to pass arguments to the child program. Generally, modern programs don't use parameters passed in the FCBs. If you have an application in which you know something about the child program (for example, that it will always be a C program), you may be able to skip preparing FCBs. However, a general-purpose debugger such as Monitor must emulate DOS start-up conditions quite exactly. Fortunately, documented DOS Function 29h (Parse File Name) makes the job easy.

Assume that you have a line of input from the user, with the name of the program to execute and its arguments. In Monitor, this will come from the command line. The function ParseFile, shown below, will find and prepare the filename, as well as set up any arguments.

```
ParseFile:
;Find start and end of file name
;Inputs:
;   ds:si = pointer to input string
;   cx = length of string
;Outputs:
;   ax = Starting value for ax (drive validity flags)
;   dx = File name to execute, zero-terminated

    mov bx,si       ;Save initial pointer
    call ScanB
    mov dx,si       ;Save starting address
    call FindNameEnd
    sub bx,si
    neg bx          ;Amount scanned so far
    sub cx,bx       ;Amount remaining in string
    mov bx,si       ;Save end of name--start of args
    mov di,offset DGroup:LineBuf+1
    mov [di-1],cl   ;Put length in first byte
    inc cx          ;Copy terminating CR
rep movsb           ;Copy to argument buffer
    mov si,bx       ;Restore start of args
    mov di,5CH      ;First FCB
    DOS ParseName,1 ;Parse file name, scan off blanks
    cbw             ;0FFH if invalid drive
    and al,ah       ;Make sure al is zero or one
    xchg cx,ax      ;Save return value in cl
    call FindNameEnd    ;Skip over any "\" chars
    mov di,6CH      ;Second FCB
    DOS ParseName,1 ;Parse file name, scan off blanks
    cbw             ;0FFH if invalid drive
    and ah,al       ;Make sure ah is zero or one
    mov al,cl
    mov byte ptr [bx],0 ;Zero terminate file name
    ret

FindNameEnd:
    lodsb
    cmp al," "      ;Check for blank or control char
    jbe NameEnd
    cmp al,","
    jz  NameEnd
    cmp al,";"
    jz  NameEnd
    cmp al,"/"
    jnz FindNameEnd
NameEnd:
```

```
    dec si          ;Point back at terminator
    ret

;Scan command line for next non-blank character
ScanB:
    lodsb
    cmp al," "
    jz  ScanB       ;Skip over blanks
    cmp al,9
    jz  ScanB
    dec si          ;Back up to first non-blank
EolChk:
    cmp al,13
    ret
```

The steps are as follows:

1. Skip over leading "white space"—tabs and blanks. Monitor has the subroutine ScanB for this purpose.
2. Scan ahead for the end of the filename. The subroutine FindNameEnd does it. The end of a filename is defined as the first blank, control character, comma, semicolon, or slash ("/"). DOS actually has several more characters in this list, such as brackets, equal and plus signs, and the quote character.
3. Everything from the end of the filename to the end of the line is the argument list. This is copied to a temporary buffer—in this case, Monitor's command input buffer.

Note that copying is needed only to zero-terminate the filename. Adding the zero to the end of the filename will destroy the very first character of the argument list.

4. Call the DOS Parse File Name function (INT 21h Function 29h) on the argument list. This will format the first FCB. It will also return a validity check on the drive specification, if one was present. If the drive was invalid, 0FFh will be returned; otherwise, either 0 or 1 will be returned. ParseFile uses a macro (located in DOS.INC on the accompanying disk) to make this and other DOS calls:

```
DOS ParseName, 1
```

This expands into:

```
mov ax, 2901h
INT 21h
```

When a program is executed by DOS, AX is the only register (other than the segment registers and stack) initialized to a known value. AL will contain 0 if the drive specification in the first FCB is valid, and 0FFh if it is not. Likewise, AH indicates the validity of the drive specification in the second FCB. The DOS Parse File Name function gives you almost just what you need. Adding two lines of code converts the returned value of 1 to 0:

```
cbw
and al, ah
```

5. Scan for the end of the first argument. This is necessary only because the DOS Parse File Name function does not understand full path names—it will stop on the first backslash.
6. Call DOS Parse File Name on the second argument, which will format the second FCB. Adjust the drive validity indicator as in step 4.
7. Zero-terminate the program name now that you're done processing the arguments.

The section of Monitor that handles the loading of the child program, including calling the ParseFile function, is shown below. Because this code fragment is taken out of context, some of the details may not be very clear. However, it does illustrate the general flow. The companion disks include the complete source code for Monitor, which you can review to fill in any missing links.

```
TermAddr    equ word ptr 0AH

    mov ax,cs
    mov bx,[CsSave]     ;First segment to free
    sub bx,ax           ;Compute paragraphs we're keeping
    DOS ResizeMem

    mov si,80H          ;Point to command line in PSP
    lodsb               ;Get length byte
    cbw
    xchg cx,ax          ;Put length in cx
```

```
    call ParseFile
    mov si,dx
    cmp byte ptr [si],0 ;Was there a file name?
    jz  Command
    mov [AxSave],ax
    mov [ExecFile].CommandTailSeg,ds
    mov [ExecFile].FCB1Seg,ds
    mov [ExecFile].FCB2Seg,ds
    mov bx,offset DGroup:ExecFile
    DOS Exec,1          ;Load, don't execute
    jc  NoFileLoad
    mov ax,[ExecFile].InitIp
    mov [IpSave],ax
    mov ax,[ExecFile].InitCs
    mov [CsSave],ax
    mov ax,[ExecFile].InitSp
    mov [SpSave],ax
    mov ax,[ExecFile].InitSs
    mov [SsSave],ax
    DOS GetPSP
    mov [DsSave],bx     ;DS = ES = PSP
    mov [EsSave],bx
    mov es,bx
    mov es:[TermAddr],offset DGroup:ProgTerminate
    mov es:[TermAddr+2],cs  ;Terminate address now set
    mov [TestPSP],bx
    mov bx,cs
    DOS SetPSP
Command:

NoFileLoad:
    mov si,offset DGroup:FileErrMsg
    jmp PrintAbort
```

Note that if the input line is empty, or if it has only a filename with no arguments, the ParseFile function still goes through all of the above steps. However, each step stalls on the terminating carriage return. If no program name is given, the pointer to the ASCIIZ filename returned by ParseFile is a pointer to a null string. Monitor checks for this and skips the whole program load effort under these conditions.

The drive validity indicators need to be in AL and AH when the program starts execution. This is done in Monitor by simply storing them in AxSave, which will be put in AX when a GO or TRACE command is given.

With the ExecBlock set up with pointers to the parsed output of ParseFile, you're about ready to call DOS. One remaining step that's easy to forget, however, is freeing the memory your program doesn't need. The inevitable result if you don't is that DOS returns with the "insufficient memory" error (carry flag set, AX = 8). You free unneeded memory using the DOS Resize Memory function (INT 21h Function 4Ah), applied to your current PSP. If you program in a high-level language, the run-time system will probably have already done this for you.

When DOS returns from the Load Program call, the initial stack and instruction pointer fields of the ExecBlock will have been filled in. Monitor simply copies these values to the saved images of the registers. In addition, the saved image of DS and ES are set to the PSP of the child, fulfilling the final requirement for imitating the initial conditions set by DOS.

Maintaining the Current PSP

After the DOS Load Program function has been performed, the current PSP is that of the child. This is true even when using the Load But Don't Execute subfunction, where control returns immediately to the parent. The primary significance of whose PSP is "current" relates to file I/O. (It is also significant for program termination, which we'll get to in a moment.)

The term *file I/O* is used loosely here. It includes things like getting a character from the keyboard through DOS, which is really reading from standard input. If you allow the parent to continue to run with the child's PSP, it will be using the child's file handles. If the child decides to redirect its input from a file, the parent's input will also come from the same file. If the parent opens a file, the child will have access to it through the same handle. This is all simply because you're not telling DOS that the parent is now running instead of the child.

Some applications may not need to do any kind of I/O through DOS once the child is loaded. In fact, the WINMON program presented later in this chapter falls into this category, because it uses a second monochrome display for output. Because they don't use DOS for I/O, debuggers like WINMON can be used to debug DOS itself, without incurring the wrath of DOS non-reentrancy. And because no DOS I/O is being performed, there is no reason to switch PSPs.

However, Monitor does not fall into this category: like most DOS debuggers (including DEBUG, Symdeb, and CodeView), it *does* use DOS I/O. Therefore, the undocumented DOS SetPSP function (INT 21h Function 50h) is used to change the PSP back to the parent's. Then, whenever the child is executed with a GO or

TRACE command, the PSP is set back to the child. Upon reentry to Monitor (upon hitting a breakpoint, for example), the PSP is again set to the parent. The following code shows Monitor's child execution and reentry routines, including the PSP swapping. Note how PUSH and POP instructions are used to save and restore the execution register image.

```
        dw  80H dup(?)      ;Working stack area
STACK   label   word
;Register save area
AxSave  dw  ?
BxSave  dw  ?
CxSave  dw  ?
DxSave  dw  ?
SpSave  dw  ?
BpSave  dw  ?
SiSave  dw  ?
DiSave  dw  ?
DsSave  dw  ?
EsSave  dw  ?
SsSave  dw  ?
CsSave  dw  ?
IpSave  dw  ?
FlSave  dw  ?

EXIT:
    mov bx,[TestPSP]    ;If no child, our own PSP
    DOS SetPSP          ;Switch over to child PSP
    xor ax,ax
    mov ds,ax

assume  ds:nothing,ss:nothing,es:nothing

    mov BreakPtVect,offset DGroup:BREAKFIX  ;Breakpoint interrupt
    mov BreakPtVect+2,CS
    mov StepVect,offset DGroup:REENTER      ;Single step interrupt
    mov StepVect+2,CS
    cli
    mov SP,offset DGroup:AxSave
    pop ax
    pop bx
    pop cx
    pop dx
    pop bp
    pop bp
    pop si
```

```
    pop di
    pop ds
    pop es
    pop ss
    mov sp,[SPSave]
    push [FlSave]
    push [CSSave]
    push [IPSave]
    IRET                ;Execute child

;Re-entry point from breakpoint. Need to decrement instruction
;pointer so it points to location where breakpoint actually
;occured.

BREAKFIX:
    push bp
    mov bp,sp
    dec word ptr [bp+2]
    pop bp

;Re-entry point from trace mode or interrupt during
;execution. All registers are saved so they can be
;displayed or modified.

REENTER:
    push ax
    push bx
    mov ax,ss
    mov bx,sp           ;Save stack pointer in AX,BX
    push cs
    pop ss
    mov sp,offset DGroup:CsSave
    push ax         ;Save SS
    push es
    push ds
    push di
    push si
    push bp
    push bx         ;Save SP
    push dx
    push cx
    push ss
    pop ds

assume  ds:DGroup

    mov ss,ax           ;restore user stack pointer
```

```
mov sp,bx
pop [BXSave]
pop [AXSave]
pop [IPSave]
pop [CSSave]
pop ax
and ah,0FEH
mov [FlSave],ax
mov [SPSave],sp
push ds
pop es
push ds
pop ss
mov sp,offset DGroup:AxSave
sti
cld
mov bx,cs            ;Change back to our PSP
DOS SetPSP
```

Handling Child Termination

To see what happens when the child program terminates, let's first review what goes on using the DOS EXEC function in the "normal" way. Using EXEC subfunction 0 to execute another program is much like making a subroutine call. First you set up all the arguments, and then you perform the INT 21h to invoke the DOS EXEC function. DOS fires up the program you've requested. That child program ends by performing one of the DOS termination functions, and DOS then returns from the INT 21h that invoked EXEC. Now you're back in the parent program, with execution resuming at the instruction after the INT 21h, just as it does for any other DOS call.

Let's take a closer look at how DOS returns to the right spot in the parent. When the parent calls the DOS EXEC function with INT 21h, the interrupt instruction puts the return address on the stack. DOS builds a new PSP for the child, grabs that return address from the parent's stack, and stores it in the child's PSP at offset 0Ah. When the child terminates, DOS frees up any memory and file handles the child was using, and jumps to the far address stored in the child's PSP at offset 0Ah.

Now let's look at this in the context of EXEC Subfunction 1 (Load But Don't Execute). You can think of DOS as doing all the same work as if you'd asked for Subfunction 0 (Load and Execute), right up until it's time to set the stack pointer and jump to the child. At that point, DOS instead copies the initial stack and in-

struction pointers into the ExecBlock you passed it and returns from the INT 21h with which you called it. In particular, the child's PSP has been set up in the normal way, including the terminate address at offset 0Ah. If you don't change things, the child's termination address will be set to the instruction in the parent after the INT 21h that was used to invoke the DOS EXEC function.

The resulting behavior would be quite bizarre. Imagine issuing the "GO" command in Monitor to execute the child. Eventually the child terminates, and execution returns to Monitor back in its initialization code where the EXEC function call is located!

Obviously, you need to change the termination address of the child. This is just a matter of setting a new address at offset 0AH in the child's PSP to a suitable location in the parent. Assuming you'd like to start execution at CS:ProgTerminate when the child terminates, you could use code like this:

```
;The current PSP is the child's after EXEC
     mov    ah,62H                  ;Get current PSP
     INT    21H                     ;Call DOS, bx = PSP
     mov    es,bx
;PSP segment in es
     mov    word ptr es:[0AH],offset ProgTerminate
     mov    word ptr es:[0CH],cs    ;The parent's CS
```

Monitor sets the child's termination address to code that prints a "Program terminated" message. It turns out, however, that there is an added complication when Monitor itself wants to terminate. If a child has been loaded that has not yet terminated (because it was not run to completion within the debugger), it must terminate before the parent does, so that its memory allocation and file handles can be freed. This is quite simple: just execute the DOS Terminate function (INT 21h Function 4Ch) with the current PSP set to the child's. (Note the similarity to the TSR deinstall technique used in chapter 5.) The parent picks up execution again at the termination address you set for the child, with the PSP automatically reset to the parent. A special check in Monitor prevents the "Program terminated" message from being printed in this case. The parent can now perform the DOS Terminate function for itself. This is shown in the code that follows.

```
; "Q" - Quit command

ProgTerminate:
```

```
;TestPSP = our own PSP if we are terminating the child in order to quit
;Monitor (suppress "program terminated" message).
    push cs
    pop  ds
    mov  bx,cs
    mov ax,bx
    xchg bx,[TestPSP]       ;Set to our own PSP to show no child
    cmp  ax,bx              ;Do we have a child?
    jz   JustExit
    mov  si,offset DGroup:ProgEndMsg
    jmp  PrintAbort         ;Print message, get next command line

Quit:
;We must end child first.  If no child, TestPSP = our own PSP, so we'll
;just terminate directly.  If there is a child, set TestPSP to our own PSP
;as a flag to ProgTerminate to suppress "program terminated" message.
    mov  bx,cs
    xchg bx,[TestPSP]
    DOS  SetPSP
JustExit:
    lds  dx,[NextInt15]
    DOS  SetVect,15H
    mov  ax,4c00h     ;Terminate, no error
    int  21H
```

When execution resumes in the parent upon termination of the child, a strange thing happens to the state of the registers. They are "restored" to the values they had when the DOS EXEC function returned—even SS and SP! (It is probably this type of funky behavior that kept Microsoft from documenting this subfunction.) It is safest to make no assumptions about register contents and to reset the stack upon return from the child.

Sample Program: Monitor

Monitor itself actually represents a bit of DOS history. I originally wrote it in early 1979 as a stand-alone debugger that fit into a 2KB ROM. In mid- and late-1980, when I was writing DOS 1.0, I modified Monitor to run under DOS and called it DEBUG. The only things added to it in the transition were the use of DOS for character and file I/O, and disassembly with the "U" command. The original Monitor was still important, however. Because it didn't use DOS for I/O, it could be used to debug DOS itself, and in fact was the only way to debug DOS—especially when I broke DOS badly enough so that it wouldn't boot any more!

The sample program on the companion disk is based on the original Monitor source code. I have broken it into modules to make things easier to find. The

main module is MON.ASM, and it includes all code related to loading the child program. TRACE.ASM includes the GO, TRACE, and REGISTER commands. UTIL.ASM has utilities such as command parsing and output formatting. DOSIO.ASM has the character I/O functions. The main body of commands of Monitor, such as DUMP, ENTER, MOVE, etc., are in COMMANDS.ASM. Finally, DIS.ASM has a newly written disassembler for the UNASSEMBLE command.

Debuggers and Windows Memory Movement

Microsoft Windows presents a new challenge to a debugger or any other tool that attempts to observe program execution. In Windows versions 2.x and in the "Real" mode of Windows 3.x, code segments are not fixed in a specific location in memory. Rather, they are loaded when needed ("on demand") and discarded when they fall into disuse. After loading, Windows will freely move segments around in an effort to make best use of available memory. All of this memory movement activity renders an ordinary debugger useless.

By the way, in Windows 3.x protected modes—what Microsoft calls "Standard" mode and "Enhanced 386" mode—this memory movement still takes place, but you don't see it. The values loaded into the segment registers are not paragraph addresses (which would change as memory moves), but protected-mode selectors, which are constant. A selector is an index into a table where the actual address of the memory block is stored. The table look-up is handled automatically by the 286/386/486 chip, so you never know it happens. It is a feature provided by Intel, rather than by Microsoft or IBM. As a result, memory addresses appear fixed, because the selector never changes. However, the mere fact that the CPU is operating in protected mode is a whole new challenge for a debugger—one that unfortunately cannot be tackled here. This discussion is limited to real-mode operation.

Suppose you have a Windows application called HELLO through which you'd like to single step to see how it works (or why it doesn't). Let's see what happens if you try to do this with DOS DEBUG.

Because DEBUG itself is not a Windows application, don't try to run it under Windows. Instead of trying to execute "DEBUG HELLO.EXE" from the Windows File Manager or MS-DOS Executive, you must run Windows under DEBUG, starting with the command "DEBUG WIN.COM HELLO".

So now you're at the DEBUG prompt, with Windows loaded and ready to run. When execution starts, Windows will see that you'd like to run HELLO.EXE.

Windows will find a chunk of memory somewhere and load the first code segment of HELLO into it. Once HELLO has started, any additional code segments it has will be loaded when they are called.

If HELLO or some other task running under Windows should need more memory, Windows may want to reorganize memory space to provide it. The code segments of HELLO might get moved around. If memory is tight, some of HELLO's code segments might even be discarded, and the memory might be allocated to some other use. If HELLO calls back into a discarded code segment, Windows will load the segment again—possibly discarding some other code segment to make room.

Staring back at the DEBUG prompt, it looks like you're stuck. The code for HELLO hasn't even been loaded yet, so how do you set your first breakpoint? Maybe there's a way—sometimes I change the PrintScreen interrupt vector (INT 5, 0000:0014) so that it points to the same place as the Breakpoint interrupt vector (INT 3, 0000:000C). Then I can start execution and push the PrintScreen key when I want to stop and get back to DEBUG. If this method works, maybe you can get your first breakpoint set in HELLO.

Suppose you set a breakpoint and continue the execution of Windows and HELLO. The code segment of HELLO in which you've set the breakpoint could get moved around. That might be all right, because the breakpoint instruction will get moved along with it. But when the breakpoint is reached, DEBUG won't be able to restore the original code, because the location doesn't correspond to a place where a breakpoint was set (DEBUG doesn't find it in its breakpoint table, in other words). Worse, if the segment is discarded, it will be reloaded from disk without the breakpoint instruction. In effect, the discard/load sequence removes your breakpoints! All in all, DEBUG looks pretty unusable in this environment.

DEBUG is not the only tool made useless by Windows' memory management. A more powerful, symbolic debugger such as SYMDEB is hit even harder, because it has no way to relate symbol names to physical addresses. Other classes of tools are similarly affected. One example is a profiler, which uses breakpoints and/or timer interrupts to determine where a program spends most of its time. Armed with this information, you know exactly where to target your efforts to optimize the program. Without a mapping from physical addresses to symbol names (or line numbers), there is no way to interpret the profiler's results. Another example tool—and one made equally useless by Windows real-mode memory management—is a test coverage analyzer, which reports what lines of code

are *not* executed when you run a program through its test suite. This shows you what areas need additional tests to cause all lines to be executed, a minimum standard for thorough testing.

The Windows SEGDEBUG Interface

Fortunately, real-mode Windows provides a mechanism to inform a debugger, profiler, or other tool about memory movement. The general scheme starts when Windows recognizes a signature in the debugger. From there, Windows is able to determine an entry point to call into the debugger to pass messages. These messages include segment load, move, and discard information, which is sufficient to allow the debugger to track any specific program location.

Windows looks for the signature as follows: The value stored at absolute address 0000:000E is fetched. This address is the high word (segment) of INT 3, the breakpoint vector. Presumably the debugger has set this vector so that it can use breakpoints; it is the *segment* of the debugger's breakpoint trap routine that Windows needs.

Windows looks at offset 100H in that segment. At that location it must find the ASCIIZ signature string "SEGDEBUG". If that string is present, the entry point to the debugger for messages from Windows is five bytes in front of the signature string, at offset 0FBh. This five-byte gap is intended to contain a far jump to the actual location in the debugger that will process Windows messages. Here's an example:

```
0000:0000 xxxx:yyyy        ;Vector 0 (divide overflow)
0000:0004 xxxx:yyyy        ;Vector 1 (single step)
0000:0008 xxxx:yyyy        ;Vector 2 (non-maskable interrupt)
0000:000C 1234:5678        ;Vector 3 (breakpoint)
0000:0010 xxxx:yyyy        ;Vector 4 (overflow [INTO])
...
1234:00FB jmp far ptr WndMessage
1234:0100 db  "SEGDEBUG",0
...
1234:5678 ;Breakpoint trap code
```

Note that the debugger has already set interrupt vector 3 to point to its breakpoint trap handler at 1234:5678. Windows will fetch the segment of the handler from the interrupt table (1234h). Then at 1234:0100, Windows finds the

SEGDEBUG signature, which has the entry point just in front of it. The code in the debugger to set this up might look something like this:

```
;Debugger's initialization code
     push    cs
     pop     ds                  ;Make sure ds = cs
     mov     dx,offset Break     ;Address of breakpoint trap
     mov     al,3                ;Set interrupt vector 3
     mov     ah,25H              ;DOS set interrupt vector fcn.
     int     21H
     ...
     org     0FBH
     jmp     WndMessage          ;Handle messages from Windows
     org     100H                ;In case JMP didn't take 5 bytes
     db      "SEGDEBUG",0
```

If the debugger is a .COM file, location 100h is special for a completely different reason: it is the first byte of the program file, and the entry point of the program. In this case, obviously, the signature will have to be moved into place. In the case of an .EXE file, the signature can be located statically. However, it can be a bit of a nuisance to find a small chunk of code with which to fill the first 0FBh bytes, rather than letting those bytes go to waste.

Messages from Windows

After the entry point is installed using the SEGDEBUG mechanism, Windows calls the entry point whenever a segment is loaded, moved, or discarded. The C calling convention is used, with the first argument (a word) indicating the type of message. The number of additional arguments and their meaning are dependent on the message type. The three messages needed for a debugger are:

- 0 Loaded segment
  - far pointer to ASCIIZ segment name (dword)
  - segment ordinal (word)
  - segment value (word)
  - program instance (word)
- 1 Moved segment
  - original segment value (word)
  - new segment value (word)

- 2 Discarded segment
  segment value (word)

Or, to put them in the format similar to a C function declaration:

```
void LoadSegmentMsg(0, char far *ModuleName, short Ordinal,
     short SegVal, short Instance, short SegType);

void MoveSegmentMsg(1, short OldSegVal, short NewSegVal);

void DiscardSegmentMsg(2, short SegVal);
```

An individual segment in Windows is identified by the name of the module it's in, along with the segment ordinal. The segment ordinal is a sequential numbering of the segments assigned by the linker. The numbering appears in the map file, although the ordinals there are one larger than those used by Windows. Ordinal zero is used in the map file to indicate "imported" functions—that is, calls to Windows; the first ordinal in the program is listed as ordinal one. Windows starts numbering the program's segments at zero, so the ordinal passed to LoadSegmentMsg is one less than that shown in the map file.

Thus, the first two arguments to LoadSegmentMsg (after the message type), ModuleName and Ordinal, identify which segment is being loaded. The next argument, SegVal, is the segment address at which it is being loaded. If more than one instance of the program is running, the Instance argument identifies the instance to which the segment belongs. (Note, however, that code is normally shared by all instances.)

The two other messages from Windows, MoveSegmentMsg and DiscardSegmentMsg, should be fairly self-explanatory. MoveSegmentMsg can be used to update any of the debugger's tables (such as breakpoint tables) that have segment values. Given OldSegVal to look for in the tables, all occurrences should be changed to NewSegVal to reflect the relocation by Windows. Similarly, DiscardSegmentMsg can be used to mark a table entry as "not present," the same state it would have before receiving LoadSegmentMsg.

Let's consider a specific sequence of events for a Windows debugger. Suppose you've just started the debugger, and WIN.COM has been loaded but not yet executed. The first thing you'll want to do is set a breakpoint in the program you're trying to debug. Of course, no code for that program has been loaded yet. The debugger must record your request for a breakpoint in terms of the segment

ordinal and offset, without actual placing the breakpoint opcode anywhere. Then you start execution of Windows with the debugger's GO command.

Using Windows, execute the program you're debugging. The debugger will be getting oodles of messages from Windows, most of which aren't relevant. Whenever it sees a LoadSegmentMsg with the same module name as the program, however, it will sit up and take note. If the segment ordinal is the same as the one for the breakpoint you wanted set, the debugger will finally have something to do. The debugger must set the breakpoint in the usual way, swapping the INT 3 breakpoint opcode with the code byte at the (now known) segment and offset you specified. The breakpoint table must be updated to include the segment value and the original code byte.

But let's assume that the breakpoint is not encountered right away, and your program continues to run. As program execution demands memory for additional code segments or other resources, the segment with your breakpoint may get moved. The debugger will get a message if this happens, and it must update the breakpoint table with the new segment value. If the program starts spending its time in other code segments, the segment of your breakpoint could be discarded. Now you're back where you started. Note that the breakpoint table had to keep the segment ordinal, even after the segment was loaded and the segment value established, just in case the segment was discarded.

At some point the debugger will see the LoadSegmentMsg again and will repeat the steps for setting the breakpoint. This time, suppose the breakpoint is reached before the segment is discarded again. As usual, the debugger will use the return address of the breakpoint opcode in searching the breakpoint table for a match. Because the segments in the table were kept updated, the debugger will find the breakpoint address in the table even if it has moved since it was set. The debugger must remove the breakpoint opcode and restore the original code byte.

Now at the debugger's command prompt, you may wish to view code or data in other segments of the program—if they're loaded. The only way for the debugger to know about those other segments, however, is from LoadSegmentMsg. That means the debugger must watch all messages for loading segments of the program, recording what goes where. In other words, it must build a table relating all segment ordinals to their current segment value. This also means paying attention to all MoveSegmentMsg and DiscardSegmentMsg to keep this table updated. That way, you can ask for a memory dump or disassembly of any seg-

ment in the program and the debugger will know what segment value to use if it's present.

Sample Program: Reporting Windows Messages

To demonstrate using the Windows SEGDEBUG interface, Monitor has been expanded to report Windows memory movement messages. Monitor doesn't use the information to track breakpoints in shifting code segments though. Rather, this sample determines which message Windows is sending and reports it along with its arguments:

```
WinMsg 13 End focus DGroup 5667 Id 04
WinMsg 14 Start focus DGroup 5415 Id 05
WinMsg 2 Discard segment 5415
WinMsg 2 Discard segment 5405
WinMsg 2 Discard segment 557B
WinMsg 2 Discard segment EC23
WinMsg 12 End program Id 05
WinMsg 14 Start focus DGroup 633FId 04
WinMsg 13 End focus DGroup 633FId 04
WinMsg 14 Start focus DGroup 5667 Id 04
```

The tasks required to put this information to use as outlined earlier would be meaningful only in a symbolic debugger unlike Monitor.

The first change made to Monitor is to use different character I/O. Monitor had been using DOS to receive characters from standard input and display characters to standard output. This method is not acceptable once Windows is running. Instead, the source file DOSIO.ASM has been replaced with MONOIO.ASM, which uses a monochrome display (MDA) for output. Using Monitor in this way requires that two display cards be present in the computer, one for Windows and one for Monitor. This is a common debugging configuration for Windows, although serial I/O is also used sometimes for the debugger. The character output code directly accesses the video display buffer, but it has been kept as simple as possible.

The only other change is the addition of the source file WINDBG.ASM. Because Monitor is a .COM file, the SEGDEBUG signature must be moved into address 100H (the entry point for .COM files) during initialization, instead of being statically allocated. Monitor uses a special segment for initialization code that allows each source file to have its own initialization code without requiring any

changes or switches in other modules. Both WINDBG.ASM and MONOIO.ASM use this feature. Each puts their initialization code in InitSeg, which is a byte-aligned public segment. At link time, all the code from each module's InitSeg is combined. Another segment called LastSeg immediately follows InitSeg; its sole contents is a RET instruction. As it starts up, Monitor makes a subroutine call to InitSeg. The initialization code from each module falls into the code for the next, until the code from the last module falls into the RET instruction in LastSeg. MONOIO.ASM clears the monochrome display screen during its initialization. WINDGB.ASM moves the SEGDEBUG signature into place.

The C calling convention used to send messages to the debugger passes the arguments on the processor stack. Arguments are pushed in reverse order of their appearance in the declaration as given above, which results in the first argument, the message type, always being pushed last. This puts the message type closest to the top of the stack, at a fixed offset where it can be found regardless of the number of additional arguments.

The C calling convention also specifies that the arguments are to be left on the stack when the procedure returns. This is very important, because it means that the debugger doesn't have to know how many arguments each message has. The debugger can be selective, processing only those messages it needs. At the same time, later versions of Windows can add new messages—or even additional arguments to existing messages—without fear of breaking existing code.

To simplify accessing the arguments in assembly language, WINMON uses the macro file CMACROS.INC from the Windows Software Development Kit (SDK). The macro for defining a word parameter, ParmW, is used to define parameters for LoadSegmentMsg, which has more parameters than any other message. The first two parameters after the message type have generic names, because they are used with different meanings by several of the messages.

The rest of WINDBG.ASM is very straightforward. The message type is used to index into a table of addresses, dispatching to a handler for that specific message. Each of the handlers simply fetches the arguments and displays them on the debugging screen. It is in these message handling routines that code must be placed to maintain the debugger's internal ordinal-to-segment mapping table.

Additional Message Types

By looking at what Windows and SymDeb do with messages, I have gained a sketchy idea of some of the other Windows messages:

```
EchoMsg(4, char far * String);
```

SymDeb displays the string on the debugging screen. I have not seen this one issued by Windows.

```
StartProgMsg(11, short ProgOrdinal)
```

This is called whenever a program is started. ProgOrdinal is a number assigned to the program, used again with EndProgMsg. When a second instance of a program is started, ProgOrdinal will be the same as the first instance.

```
EndProgMsg(12, short ProgOrdinal)
```

This is called when a program has terminated. ProgOrdinal is the same as returned by StartProgMsg.

```
EndDgroupMsg(13, short SegVal, short ProgOrdinal)
```

Always followed by StartDgroupMsg, the SegVal is the segment that has been (but no longer will be) the default data segment. This is called very often—every time the input focus changes.

```
StartDgroupMsg(14, short SegVal, short ProgOrdinal)
```

SegVal is the new default data segment; see EndDgroupMsg. EndDgroup Msg is called whenever a program loses the input focus; StartDgroupMsg whenever a program gains the input focus.

Conclusion

This chapter has discussed two different debugger interfaces available under MS-DOS. Neither interface is documented or supported by Microsoft, but the first interface (INT 21h Function 4B01h) is essential to any programmer developing a "normal" DOS debugger, and the second interface (SEGDEBUG) is essential for anyone writing a debugger for real-mode Windows. These interfaces can even be important for a wider class of programs than just debuggers. For example, execution profilers can also use INT 21h Function 4B01h. Perhaps a character-based

non-Windows application could use SEGDEBUG as part of communication with Windows applications.

On the other hand, this chapter did not explore the area of protected-mode debugging, either for DOS or for Windows. In Windows 3.0 "Standard" and "Enhanced" modes, developers can use the protected-mode CVW debugger, whose operation is completely different from SEGDEBUG. Even under DOS or Windows, protected-mode debugging requires an interface more like that of OS/2's DosPTrace() function. This function, interestingly enough, is largely undocumented: it seems that for those writing debuggers and other diagnostic tools, use of undocumented operating system features will have to carry over into the world of protected mode as well.

Chapter 8

INTRSPY: A Program for Exploring DOS

David Maxey

Several times in this book we have referred to the program INTRSPY. In chapter 1, we used it to see which programs use undocumented DOS. In chapter 4, we used it to explore the workings of the MS-DOS network redirector. In chapter 6, we referred to an INTRSPY script that helped us figure out how the INT 2Fh Function AEh (Installable Command) interface works.

It is now time to examine INTRSPY in detail. After explaining how INTRSPY differs from other debuggers, this chapter presents a quick sample session with the program. This should be sufficient to get you started using INTRSPY; the program itself is on the disks that accompany this book. After this "guided tour," a more formal "user's guide" to the program is presented, followed by an examination of several sample scripts.

The second half of this chapter discusses the specification for INTRSPY, key design issues, and the program's implementation in Turbo Pascal.

Why a Script-Driven, Event-Driven Debugger?

INTRSPY is an event-driven debugger: it takes over one or more interrupt vectors and, when the interrupt is generated, performs some action. This sets it apart

from more conventional debuggers, such as DEBUG, CodeView, or Turbo Debugger, which are generally "driven" by a user's keystrokes.

There are already several other DOS debuggers that intercept interrupts, allowing you to look into the DOS and/or BIOS activity on your PC. JPI TopSpeed C comes with an excellent program called WATCH, which monitors DOS (INT 21h) calls. IBM used to market a program called PCWATCH in its now-defunct "Personally Developed Software" series; this program allowed you to monitor any software interrupt, not just INT 21h calls.

Such a program is essential, not only for exploring undocumented DOS, but for many other debugging tasks on the PC as well. For example, what do you do if a program exits unexpectedly because it can't find some configuration file, but the program's error messages don't tell you *which* file it couldn't find? Just intercept INT 21h and monitor the different functions involved with opening, creating, or finding files.

However, INTRSPY is different from existing DOS monitoring programs like JPI WATCH, because it provides a scripting *language* for intercepting interrupts: it is event-driven *and* script-driven. The INTRSPY program knows very little about any particular DOS or BIOS call. It doesn't know that INT 21h Function 3Dh is the Open File function; it doesn't know that this function takes the ASCIIZ path name of a file to open to in the DS:DX register pair, or that, if successful, the function returns a file handle in the AX register. Rather than hard-wire such "protocol" knowledge into the program, INTRSPY's scripting language lets *the user* provide such knowledge.

The benefit is that the program is open-ended. If you want to monitor some undocumented region of DOS that this book doesn't mention, you can. If you want to examine some little-known DOS subsystem, you just write a script. Furthermore, because the INTRSPY language includes support for *strings* and *structure*, you can produce meaningful output rather than raw register dumps.

A Guided Tour

Let's first run through a quick session with INTRSPY. It makes sense to start with something that *doesn't* involve undocumented DOS, so pretend you are interested in tracking down which files a program opens. You could, of course, disassemble the program, or run it under a debugger, but it makes more sense to treat the program as a "black box" and simply see what DOS file calls it makes. In

other words, you should study the program as if you were a behavioral, rather than a Freudian, psychologist.

INTRSPY is perfect for this sort of exploration. You could, for example, monitor the file activity required to compile INTRSPY itself. However, INTRSPY (as you will see shortly) is written in Turbo Pascal, and part of the reason for TP's blazingly fast compilation is that it doesn't create temporary files or spawn subprocesses. So, let's instead look at a compiler with a somewhat more baroque file usage. The following commands could be used to find out what files Microsoft C 6.0 uses when compiling a tiny HELLO.C program:

```
intrspy -r10240
cmdspy compile fopen.scr
cl hello.c
cmdspy report
```

This code first loads INTRSPY, which is a memory-resident program. It then uses CMDSPY to compile an INTRSPY script called FOPEN.SCR. As explained below, CMDSPY communicates with the resident INTRSPY program. Then the Microsoft CL program is run. Finally, the code produces a report (which we also could have sent directly to a file with a command such as CMDSPY REPORT CL.LOG).

What does FOPEN.SCR look like? Here is a very simple version, which only traps calls to INT 21h Function 3Dh (Open File):

```
; FOPEN.SCR (simple version)
intercept 21h
    function 3dh     ; Open File
        on_entry
            output "OPEN " (ds:dx->byte,asciiz,64)
        on_exit if (cflag == 1)
            sameline " [FAIL " ax "]"
```

This script instructs INTRSPY to intercept INT 21h and trap all calls with AH=3dh (Function 3Dh). On entry to the call, INTRSPY should output the string "OPEN ", plus the ASCIIZ string pointed at by the DS:DX register pair; a maximum of 64 bytes will be stored. (Here, the -> operator indicates how the memory DS:DX points to should be formatted.) On exit, if the carry flag is set, the code tells INTRSPY it should output (on the same line) the string "FAIL" and the value of the AX register, enclosed in square brackets.

Note how the on_entry clause corresponds to the parameters expected by INT 21h Function 3Dh, and how the on_exit clause corresponds to its possible return values:

```
INT 21h Function 3Dh
Open File
Call with:
    AH = 3Dh
    AL = access mode
    DS:DX -> segment:offset of ASCIIZ pathname
Returns:
    CARRY = clear if function successful
        AX = handle
    CARRY = set if function unsuccessful
        AX = error code
```

The output from INTRSPY goes, not directly to your screen, but into a results buffer. The default buffer size is 2KB; we created a 10KB buffer here by specifying INTRSPY -r10240.

The command CMDSPY REPORT produced the following results after running CL HELLO.C:

```
OPEN c1.exe [FAIL 0002]
OPEN c:\tmp\004990sy [FAIL 0002]
OPEN c:\tmp\004990sy
OPEN c:\tmp\004990ex [FAIL 0002]
OPEN c:\tmp\004990ex
OPEN c:\tmp\004990in [FAIL 0002]
OPEN c:\tmp\004990in
OPEN c:\tmp\004990st [FAIL 0002]
OPEN c:\tmp\004990st
OPEN hello.c
OPEN c:/msc/include\stdio.h
OPEN c2.exe [FAIL 0002]
OPEN c:\tmp\004990ex
OPEN c:\tmp\004990sy
OPEN c:\tmp\004990in
OPEN c:\tmp\004990pr [FAIL 0002]
OPEN c:\tmp\004990pr
OPEN c:\tmp\004990gs [FAIL 0002]
OPEN c:\tmp\004990gs
```

```
OPEN c:\tmp\004990ls [FAIL 0002]
OPEN c:\tmp\004990ls
OPEN C:\MSC\BIN\c2.exe
OPEN c3.exe [FAIL 0002]
OPEN c:\tmp\004990pr
OPEN c:\tmp\004990gs
OPEN c:\tmp\004990ls
OPEN c:\tmp\004990in
OPEN c:\tmp\004990st
OPEN hello.obj
OPEN c:\tmp\004990lk [FAIL 0002]
OPEN link.exe [FAIL 0002]
OPEN C:\BIN\link.exe
OPEN c:\tmp\004990lk
OPEN hello.obj
OPEN c:\msc\lib\SLIBCE.lib
OPEN C:hello.exe
OPEN C:hello.exe
OPEN FOPEN1.SCR
```

Whew: All that just to compile HELLO.C! The temporary files with sy, ex, gs, ls, in, st, pr, and lk suffixes presumably relate to different aspects of compiling: symbols, expressions, global optimization, local optimization, etc. If you have Microsoft C 6.0, you might try the new -qc option to do a quick compile. Instead of the barrage of file activity you get with a full optimizing compile, the INTRSPY results then look like this:

```
OPEN qcc.exe [FAIL 0002]
OPEN hello.obj
OPEN hello.c
OPEN c:/msc/include\stdio.h
OPEN c:\tmp\004990lk [FAIL 0002]
OPEN link.exe [FAIL 0002]
OPEN C:\BIN\link.exe
OPEN c:\tmp\004990lk
OPEN hello.obj
OPEN c:\msc\lib\SLIBCE.lib
OPEN C:hello.exe
OPEN C:hello.exe
OPEN FOPEN1.SCR
```

No wonder the -qc switch is faster.

What have you accomplished here? With a seven-line INTRSPY script, you have created a file-open logging utility that would have taken many more lines of code (and, more important, a good several hours of programmer's time) to write (and debug!) in C, assembly, or Pascal language.

Still, the FOPEN.SCR file is only a minimal implementation of a file-logging utility. For example, simply by typing CL, you must be generating lots of file system activity as DOS first tries to find, and then tries to execute, CL.EXE. Furthermore, your log doesn't show any files being *created.*

All you need to do is trap some additional functions, as in this next beefed-up version of FOPEN.SCR. If you want to see the DOS command-line with which programs are EXECed, we also need to provide an INTRSPY STRUCTURE that represents the parameter block used by the DOS EXEC function:

```
; FOPEN.SCR
structure param_blk fields
    env_seg (word,hex)
    args (dword,ptr)

intercept 21h
    function 3ch    ; Create File
        on_entry
            output "CREAT " (ds:dx->byte,asciiz,64)
        on_exit if (cflag == 1)
            sameline " [FAIL " ax "]"
    ; ---------------------------------------
    function 3dh    ; Open File
        on_entry
            output "OPEN  " (ds:dx->byte,asciiz,64)
        on_exit if (cflag == 1)
            sameline " [FAIL " ax "]"
    ; ---------------------------------------
    function 4bh    ; Execute Program
        subfunction 00h
            on_entry
                output "EXEC  "
                       (ds:dx->byte,asciiz,64)                  ; program
                       (es:bx->param_blk.args->byte,string,64) ; cmdline
            on_exit if (cflag == 1)
                sameline " [FAIL " ax "]"
    ; ---------------------------------------
    function 4eh    ; Find First File
        on_entry
            output "FIND  " (ds:dx->byte,asciiz,64)
```

```
on_exit if (cflag == 1)
    sameline " [FAIL " ax "]"
```

The INTRSPY STRUCTURE statement corresponds to the first two fields (the only ones you're interested in here) of the parameter block that INT 21h Function 4Bh Subfunction 00h expects from the ES:BX register pair. You output the DOS command line (or the first 64 bytes of it) with this expression:

```
(es:bx->param_blk.args->byte,string,64)
```

This indicates that the ES:BX pair points to (->) a param_blk structure, and that you are interested in the args field. We in turn use another -> to indicate that args (dword,ptr) points to a string. A STRING designation is different from ASCIIZ, in that its first byte is a length count. This corresponds exactly with the command tail used by MS-DOS.

Now you get even more output describing the activity generated by the simple command CL HELLO.C. There's too much to show it all, but here are the highlights:

```
FIND  cl.???
FIND  C:\QEMM\cl.??? [FAIL 0012]
FIND  C:\TURBO\cl.??? [FAIL 0012]
FIND  C:\MSC\BINB\cl.??? [FAIL 0012]
FIND  C:\BIN\cl.??? [FAIL 0012]
FIND  C:\EPS\cl.??? [FAIL 0012]
FIND  C:\MSC\BIN\cl.???
EXEC  C:\MSC\BIN\CL.EXE hello.c
OPEN  c1.exe [FAIL 0002]
EXEC  C:\MSC\BIN\c1.exe
OPEN  c:\tmp\005116sy [FAIL 0002]
CREAT c:\tmp\005116sy
    .
    .
    .
OPEN  hello.c
OPEN  c:/msc/include\stdio.h
OPEN  c2.exe [FAIL 0002]
EXEC  C:\MSC\BIN\c2.exe
    .
EXEC  C:\MSC\BIN\c3.exe
```

```
OPEN  c:\tmp\005116lk [FAIL 0002]
CREAT c:\tmp\005116lk
OPEN  link.exe [FAIL 0002]
EXEC  C:\BIN\link.exe @"\"c:\tmp\005116lk\"
OPEN  C:\BIN\link.exe
OPEN  c:\tmp\005116lk
FIND  c:\tmp\005116lk
OPEN  hello.obj
FIND  hello.obj
OPEN  c:\msc\lib\SLIBCE.lib
FIND  c:\msc\lib\SLIBCE.lib
OPEN  C:hello.exe
CREAT C:hello.exe
    .
    .
    .
```

The first thing you see here is that, before the actual execution of CL.EXE itself, COMMAND.COM must first *find* it. The series of failed calls to the DOS Find First function show COMMAND.COM looking along the PATH in many subdirectories before it finally finds CL.EXE. If you were going to be using Microsoft C a lot, it would probably be a good idea to optimize your PATH by moving C:\MSC\BIN forward a little.

You can also see the invocation of the separate executables that comprise Microsoft C: C1.EXE, C2.EXE, and C3.EXE. There are no command-line arguments here because these programs communicate among themselves using the MSC_CMD_FLAGS environment variable, plus all those temporary files.

You have seen that INTRSPY can be likened to a "protocol analyzer" for PC software interrupts, where INTRSPY itself only knows about raw interrupts, registers, and interrupts, and where the user's scripts impose the necessary higher-level interpretation to understand what is actually going on.

INTRSPY User's Guide

INTRSPY is actually two programs, INTRSPY.EXE and CMDSPY.EXE, whose operating instructions follow.

Usage of INTRSPY.EXE The following command is used to run INTRSPY.EXE:

```
INTRSPY [-rnnnn] [-innnn]
```

where:

- -rnnnn specifies the amount of memory INTRSPY is to allocate for result storage (default is 2KB).
- -innnn specifies the amount of memory INTRSPY is to allocate for interrupt handler code (default is 1KB).

For example, the following commands runs INTRSPY with an allocation of 24,000 bytes of results space and the default 1KB of interrupt handler code space:

```
C:\>INTRSPY -r24000
```

Usage of CMDSPY.EXE The following command is used to run CMDSPY.EXE:

```
CMDSPY  [COMPILE [d:][path]inptfile[.ext] [param-1 [param-2 .. ]]]
        [REPORT [[d:][path]outpfile.ext]]
        [RESTART]
        [FLUSH]
        [STOP]
        [UNLOAD]
```

where:

- COMPILE compiles a script, and instructs INTRSPY to begin monitoring interrupts and storing results specified in inptfile.ext, which contains script source in the form defined below. Any currently active script is stopped, and the results area is flushed. If the extension is omitted, .SCR (script) is assumed. For example, the following DOS command line compiles the script TEST.SCR:

```
C:\>CMDSPY COMPILE TEST
```

- REPORT instructs INTRSPY to return the results accumulated so far. The file outpfile.ext, if specified, will contain the formatted results of the current script since the last time it was processed, unless there has been an intervening FLUSH.
- STOP instructs INTRSPY to stop monitoring interrupts but to preserve the results area.
- RESTART instructs INTRSPY to restart monitoring interrupts (after a STOP command) on the basis of the currently compiled script.

- FLUSH instructs INTRSPY to clear the results area but to leave the currently script active.
- UNLOAD instructs the INTRSPY TSR to unload itself from memory. Any active script is stopped.

Script Language

The script language allows eight main constructs. These are:

- INCLUDE, which includes another input file.
- STRUCTURE, which defines a data structure.
- INTERCEPT, which specifies an interrupt, optional function, and optional subfunctions, together with the entry and exit processing to be done when that interrupt is triggered.
- RUN, which allows a DOS program to be EXECed from within the script
- REPORT, FLUSH, STOP, and RESTART, all of which work exactly like their command-line counterparts described above.

Syntax

A script file is an ASCII file. All white space is ignored, except within literal strings used for results output. Thus, indentation and multiple lines may be used for readability. Line endings are only used to delimit comments, which begin with a semicolon anywhere on a line.

The placeholders %1 through %9 can appear anywhere in a script and are replaceable from the DOS command line. The following is a valid INTRSPY script:

```
; INTERCEPT.SCR
intercept %1
    function %2
        %3 %4 %5 %6 %7 %8 %9
```

This script could be used for one-shot queries that didn't deserve their own separate scripts. For example:

```
C:\>cmdspy compile intercept 21h 52h on_exit output es ":" bx
```

The simplest possible (though degenerate) INTRSPY script is thus:

```
; SCRIPT.SCR
%1 %2 %3 %4 %5 %6 %7 %8 %9
```

which then requires that the entire script be placed on the DOS command line:

```
C:\>cmdspy compile script intercept 21h function 52h on_exit output es ":" bx
```

INCLUDE Syntax The following syntax is used for INCLUDE statements:

```
INCLUDE "[d:][path]inptfile[.ext] [param-1 [param-2 ...]]"
```

This syntax includes the specified file and substitutes param-1, param-2, etc., in the source in place of the strings %1, %2, etc., respectively, where found. If the extension is omitted, .SCR is assumed.

STRUCTURE Syntax Let us define a field definition as:

```
field-type [,field-disp-type [,field-dup]]
```

where:

`field-type` can be `BYTE`, `WORD` or `DWORD`.

- field-disp-type can be HEX, BIN, DEC, PTR, STRING, ASCII, ASCIIZ, or DUMP (a combination of HEX and ASCII).
- field-dup is the number of elements if the field is an array, or the length of field in, for example, a string. In the case of a field being defined within a structure definition, field-dup may be a numeric literal or one of the predefined constants. In the case of a definition within an output-element (see INTERCEPT syntax below), field-dup may also refer to a register (reg8, reg16, sreg).

Then, STRUCTURE syntax looks like this:

```
STRUCTURE struct-name FIELDS
     field-name1 (field-definition)
    [field-name2 (field-definition)]
          .
          .
    [field-nameN (field-definition)]
```

- struct-name must be a unique structure identifier.
- field-name must be unique within struct-name.

For example:

```
STRUCTURE param_blk FIELDS
    env_seg (WORD,HEX)
    args (DWORD,PTR)
```

Note that there are only 12 significant characters for both struct-name and field-name. Thus, LISTOFLISTS_30 and LISTOFLISTS_31 would *not* be unique.

INTERCEPT Syntax Let us define an output-element as:

```
REGS or
sreg-name, reg16-name, reg8-name, flag-name or
"string literal" or
(segval:ofsval[incr]->struct.field->struct.field->struct) or
(segval:ofsval[incr]->struct.field->struct.field->struct.field) or
(segval:ofsval[incr]->struct.field->struct.field->field-definition) or
predefined-constant
```

where:

- sreg-name is the name of a segment register (that is, CS, DS, ES, or SS).
- reg16-name is the name of a 16-bit register (that is, AX, BX, CX, DX, DI, SI, BP, SP, IP, CS, DS, ES, or SS).
- reg8-name is the name of an 8-bit register (that is, AH, AL, BH, BL, CH, CL, DH, or DL).
- flag-name is the name of a flag: OFLAG (overflow), DFLAG (direction), IFLAG (interrupt), TFLAG (trap), SFLAG (sign), ZFLAG (zero), AFLAG (auxiliary), PFLAG (parity), or CFLAG (carry).
- segval is an sreg-name, reg16-name, numeric literal, or predefined constant.
- ofsval is an sreg-name, reg16-name, numeric literal, or predefined constant.
- [incr] is an optional numeric literal increment.
- struct.field-> may appear up to four times, and struct and field must be predefined.
- predefined-constant is one of those defined in the section "Predefined Constants" on page 467.

Let us then define an output-clause as one of the following:

- OUTPUT output-element [output-element [output-element ...]]], which starts on a new line
- SAMELINE output-element [output-element [output-element ...]]], which attempts to append elements to an existing line
- STREAM reg8-name, which outputs raw ASCII characters from the 8-bit register specified
- DEBUG, which, if the intercept occurs during a RUN statement, enters a pop-up debugger (see below)

Let us then define a test-clause as:

- (something==something) or (something!=something), where something may be a sreg-name, reg16-name, reg8-name, flag-name, or predefined constant.

Let us next define an if-clause as follows:

```
IF test-clause [AND test-clause [OR test-clause [ ... ]]]
    [output-clauses]
```

Let us add the two keywords ON_ENTRY and ON_EXIT, to describe pre- and post-processing of an interrupt.

Finally, as shorthand for the tests IF (ah == value) and IF (al == value), let's define two additional keywords:

```
FUNCTION ah-value

SUBFUNCTION al-value
```

(Actually, these are not strictly equivalent to testing the value of the AH and AL registers, because FUNCTION and SUBFUNCTION operate not only ON_ENTRY, but also ON_EXIT, where the value in AH or AL might have been changed.)

Then, INTERCEPT syntax looks like this:

```
INTERCEPT interrupt-number
    [output-clauses]
    [test-clauses]
    [FUNCTION fnctn-number [fnctn-number ...]
```

```
        [output-clauses]
        [test-clauses]
        [SUBFUNCTION sfnctn-number [sfnctn-number ... ]
            [output-clauses]
            [test-clauses]
                [ON_ENTRY
                    [output-clauses]
                    [test-clauses]]
                [ON_EXIT
                    [output-clauses]
                    [test-clauses]]]
        [SUBFUNCTION sfnctn-number [sfnctn-number ... ]
                .
                .   ]
    [FUNCTION fnctn-number [fnctn-number ... ]
            .
            .
            .       ]
```

For example:

```
INTERCEPT 21h
    FUNCTION 3Ch    ; Create File
        ON_ENTRY
            OUTPUT "CREAT " (DS:DX->BYTE,ASCIIZ,64)
        ON_EXIT
            IF (CFLAG == 1)
                SAMELINE " [FAIL " AX "]"
```

RUN syntax RUN syntax looks like this:

```
RUN "[d:][path]program[.ext] [parm1 [parm2 ...]]"
```

If the extension is omitted, first an.EXE and then a .COM file is searched for, either in d:path if specified or on the DOS search PATH if both drive and path are omitted. Substitute param-1, param-2, etc., in the source in place of the strings %1, %2, etc., respectively, where found.

For example:

```
RUN "cl hello.c"
```

or:

```
RUN "%1 %2 %3 %4 %5 %6 %7 %8 %9"
```

If the second example were embedded in a script called TEST.SCR, the syntax from the DOS command line might then look like this:

```
C:\>CMDSPY COMPILE TEST cl hello.c
```

Because DOS programs can be run normally from the DOS command line while an INTRSPY script is active, the RUN statement is necessary only when (a) you want to investigate a program without possible interference from COMMAND.COM; (b) where you want an entirely self-contained script; or (c) you want to use the DEBUG statement (see below).

REPORT syntax REPORT syntax looks like this:

```
REPORT "[d:][path]outpfile.ext" or "" for STDOUT
```

STOP and RESTART syntax Neither STOP nor RESTART take any parameters.

DEBUG syntax DEBUG syntax looks like this:

```
DEBUG "message" or ""
```

If its associated intercept occurs during a RUN statement, the DEBUG statement invokes a simple interactive debugger that allows access to some of CMDSPY's output capabilities from a command line. The DEBUG statement takes one parameter: a string to display in the debugger.

Debugger commands are the following:

- R

This displays the registers as they were when the caller generated the interrupt, but also reflecting any modifications made using the M (Modify) command shown below. It replicates the REGS output statement.

- D

```
segval:ofsval[incr]->struct.field->struct.field->field-definition
```

This displays an area of memory. The argument to the command is an output-element, but without the parentheses.

- M register-name = new-value

This modifies the contents of a register (sreg, reg16, orreg8) to be new-value. new-value may be an sreg-name, reg16-name, reg8-name, numeric literal, or predefined constant.

- C

This cancels all register modifications and returns all register contents to their values at entry.

- X

This exits the debugger. If modifications have been made, it allows the modifications to be canceled or allowed to remain in effect.

- ↑,↓

The Up and Down arrow keys recall previous commands for editing.

The command line is fully editable using the normal editing keys. <ESC> clears the command line, which is permanently in Insert mode.

An INTRSPY script that invokes the debugger might look like this:

```
; OPEN.SCR
intercept 21h
    function 3dh
        on_entry
            debug "INT 21h Function 3Dh - Open File"

run "%1 %2 %3 %4 %5 %6 %7 %8 %9"
```

Note that the debugger is available only within a script that uses a RUN statement. As noted earlier, a RUN statement can contain either a string literal (for example, RUN "CL HELLO.C") or parameters replaceable from the DOS command line. In this example, you might then type the following at the DOS prompt:

```
C:\>cmdspy compile open cl hello.c
```

At the first call to INT 21h Function 3Dh, you would be in the debugger:

```
Reg  AX   BX   CX   DX   SI   DI   DS   ES   BP   SS   SP   CS   IP    FLAGS
Val 3D00 371E 0000 0437 33F0 0BE0 4173 4173 3094 4173 308E 3C34 3BD9 odItsZaPc
```

```
== ON ENTRY==================================================================
INT 21h Function 3Dh - Open File
Command >>d ds:bx->byte,asciiz,64

Data:C:\MSC\BIN\c1.err

== ON ENTRY==================================================================

Command >>x
```

Note that at the debugger >> prompt you can type in expressions similar to those enclosed in INTRSPY scripts.

As noted earlier, the DEBUG statement invokes only the debugger when located in a script with a RUN statement; without RUN, DEBUG is a NOP.

Predefined Constants The following constants are provided within the script language and the debugger:

- OS_MAJOR and OS_MINOR

These constants contain the major and minor DOS version number, obtained from INT 21h Function 30h (Get Version).

- LOL_SEG and LOL_OFS

These constants contain the segment and offset portions of the address of the DOS list of lists, obtained from INT 21h Function 52h (Get List of Lists).

- SDA_SEG and SDA_OFS

These contents contain the segment and offset portions of the address of the primary Swappable DOS Area, obtained from INT 21h Function 5D06h (Get Swappable DOS Area). (This is useful in exploring the network redirector.)

Error Messages

The following list describes some possible error messages from INTRSPY and CMDSPY, as well as suggested corrective actions:

INTRSPY already loaded...
Meaning: You are loading INTRSPY while a previous invocation of it is resident.
Action: If you need to change the current handler or result space allocations, use CMDSPY UNLOAD to remove the currently loaded copy from memory.

INTRSPY vN.NN not loaded....
Meaning: You are running CMDSPY and it is unable to locate INTRSPY in memory.
Action: Load INTRSPY and retry.

Insufficient memory...
Meaning: You are loading INTRSPY and there is not enough memory for it to allocate any heap memory, or you are running CMDSPY and it is unable to allocate enough memory for its needs.
Action: Ensure that you have at least 64KB DOS memory available.

Error opening file
Meaning: CMDSPY cannot locate the specified script file.
Action: Check that the file exists and that its name is correctly spelled.

Bad switch -> xxxxxxx
Meaning: You are loading INTRSPY with an unrecognized command-line switch.
Action: Rerun INTRSPY with a corrected command line.

Bad result space switch -> xxxxx
Meaning: You are loading INTRSPY but have specified -rxxxxx where xxxxx is non-numeric or not in the required range (1–65520).
Action: Rerun INTRSPY with a corrected command line.

Bad handler space switch -> xxxxx
Meaning: You are loading INTRSPY but have specified -ixxxxx where xxxxx is non-numeric or not in range (1–65520).
Action: Rerun INTRSPY with a corrected command line.

INTRSPY returned: Out of handler space
Meaning: You are attempting to process the line CMDSPY COMPILE. INTRSPY has insufficient handler space left to accommodate code compiled for the current intercept.
Action: Unload INTRSPY and reload with a higher -innnn amount.

INTRSPY returned: Script active
Meaning: You are attempting to process the line CMDSPY COMPILE, STOP, or UNLOAD. CMDSPY has requested INTRSPY to disable the current script and reset intercepted interrupts to the original vectors, but INTRSPY is unable to do so because it is in the middle of interrupt processing. However, it will attempt to complete the reset as soon as the function (usually INT 21h Function 4bh EXEC) has finished.
Action: Retry the command. Reboot if unsuccessful.

INTRSPY returned: Vectors superseded
Meaning: You are attempting to process the line CMDSPY UNLOAD. CMDSPY has requested INTRSPY to disable the current script, reset intercepted interrupts to the original vectors, and unload itself from memory. INTRSPY is unable to do so because of a subsequently installed TSR.
Action: Unload any subsequently loaded TSRs and retry. Reboot if unsuccessful.

Download contains unrecognized structure.
Meaning: You are attempting to process the line to CMDSPY REPORT. CMDSPY has encountered a reference to a structure in the results downloaded from INTRSPY that is not present in the script that it has recompiled. This indicates that the script file has been edited between CMDSPY COMPILE and CMDSPY REPORT.
Action: Rerun the entire process from CMDSPY COMPILE to CMDSPY REPORT.

Download contains unrecognized string literal.
Meaning: You are attempting to process the line CMDSPY REPORT. CMDSPY has encountered a reference to a string literal in the results downloaded from INTRSPY that is not present in the script that it has recompiled. This indicates that the script file has been edited between CMDSPY COMPILE and CMDSPY REPORT.
Action: Rerun the entire process from CMDSPY COMPILE to CMDSPY REPORT.

Using INTRSPY

Let's now take a look at some mini-applications built using the INTRSPY language.

UNDOC

First, let's focus on how INTRSPY can show quickly which undocumented DOS calls a program makes. Some of this material was presented in chapter 1, but then we were interested only in the final results, not in INTRSPY itself. Here, we'll focus on just two pieces of system software—the Microsoft CD-ROM Extensions (MSCDEX) and the NetWare shell (NET3). Here is a step-by-step account:

```
C:\>intrspy                                          (1)
INTRSPY v1.00 is now resident.

C:\>cmdspy compile undoc                             (2)
Compiling UNDOC.SCR
UNDOC.SCR compiled OK.
Generated  404 bytes of code for interrupt 21h.
Generated   20 bytes of code for interrupt 2Eh.

C:\>mscdex /d:mscd0001 /l:l                          (3)

C:\>cmdspy report                                    (4)
Compiling UNDOC.SCR
UNDOC.SCR compiled OK.
INTRSPY returned a total of 167 bytes of results.
--------------------------                           (5)
 C:\BIN\MSCDEX.EXE
2152: Get List of Lists: 028E:0026
---------- TSR -----------                           (6)
 C:\INTRSPY\CMDSPY.EXE

C:\>CMDSPY flush                                     (7)

C:\>net3                                             (8)

C:\>cmdspy report                                    (9)
Compiling UNDOC.SCR
UNDOC.SCR compiled OK.
INTRSPY returned a total of 204 bytes of results.
-------------------------
 C:\NOVELL\NET3.EXE
2134: InDOS flag: 029F:02CF
```

```
2134: InDOS flag: 029F:02CF
2152: Get List of Lists: 029F:0026
2150: Set PSP: 10E6
2150: Set PSP: 1DDD
--------- TSR -----------
 C:\INTRSPY\CMDSPY.EXE
```

At (1), INTRSPY is loaded using the default allocations for interrupt-handling code and results space. At (2), UNDOC.SCR is compiled. This script is reproduced here:

```
;; ******
;; UNDOC.SCR (abridged version)
;; This does not show undocumented redirector (Int 2Fh) calls
intercept 21h
    function 1fh on_exit output "211F: Get Default DPB: " DS ":" BX
    function 32h on_entry output "2132: Get DPB: " DL
    function 34h on_exit output "2134: InDOS flag: " ES ":" BX
    function 50h on_entry output "2150: Set PSP: " BX
    function 51h on_exit output "2151: Get PSP: " BX
    function 52h on_exit output "2152: Get List of Lists: " ES ":" BX
    function 53h on_exit output "2153: Translate BPB"
    function 5dh subfunction 06h
        on_exit output "215D06: Get DOSSWAP: " DS ":" SI
    function 60h
        on_entry output "2160: Canon File: " (DS:SI->byte,asciiz,64)
        on_exit sameline " ==> " (ES:DI->byte,asciiz,64)
    function 25h
        on_entry
            if (al == 28h) output "SetVect INT 28h: KBD busy loop"
;; ******
;; Use the next functions and ints 20h and 27h to show which
;; program made the undoc DOS call, and to show termination
;;
    function 4bh
        subfunction 00h
            on_entry
                output (DS:DX->byte,asciiz,64)
        subfunction 01h
            on_entry
                output "214B01: EXEC debug: " (DS:DX->byte,asciiz,64)
    function 4ch on_entry output "-------------------------"
    function 31h on_entry output "--------- TSR -----------"
intercept 20h on_entry output "-------------------------"
intercept 27h on_entry output "--------- TSR -----------"

intercept 2eh on_entry output "2E: Execute command"
```

At (3), MSCDEX is run. At (4), a screen report of the results so far is generated. So that it is possible to distinguish which calls were generated by which program in the event that we want to accumulate results for a while, or in the event that one program spawns others, the script monitors the DOS EXEC and termination functions and interrupts. The line of hyphens at (5) is the end of the run of CMDSPY from (2) that loaded the script. The line following it shows that MSCDEX has started. At (6), it has terminated but stayed resident. In the next line, we see the invocation of CMDSPY corresponding to (4).

At (7), the result space is cleared. This isn't really necessary, but it ensures that the next report is limited to what happened in NET3, without reiterating the MSCDEX results. At (8), NET3 is run, and at (9), the results of its loading are reported.

In this experiment, we see that MSCDEX calls INT 21h Function 52h at load time, and that NET3, in addition to Function 52h, also calls Functions 34h and 50h. The reasons these programs make these particular undocumented DOS calls should be clear to you from chapters 4 and 5: the DOS List of Lists retrieved with Function 52h contains a pointer to the DOS Current Directory Structure (CDS); MSCDEX alters the CDS. Functions 34h and 50h are both important for TSRs like NET3.

LSTOFLST

In chapter 2, we went through a fairly laborious process using C to display the DOS List of Lists. It is simpler to do this with INTRSPY:

```
; LSTOFLST.SCR
; INTRSPY script to examine DOS List Of Lists (INT 21h Function 52h)

structure list_20 fields            ; DOS 2.x
    share_retry_count (word)
    retry_delay (word)
    curr_disk_buff (dword, ptr)
    unread_con (word)
    mcb (word)
    dpb (dword, ptr)
    file_tbl (dword, ptr)
    clock (dword, ptr)
    con (dword, ptr)
    num_drives (byte, hex)
    max_bytes (word)
    first_disk_buff (dword, ptr)
```

```
    nul (byte,dump,18)

structure list_30 fields            ; DOS 3.0
    share_retry_count (word)
    retry_delay (word)
    curr_disk_buff (dword, ptr)
    unread_con (word)
    mcb (word)
    dpb (dword, ptr)
    file_tbl (dword, ptr)
    clock (dword, ptr)
    con (dword, ptr)
    num_blk_dev (byte,hex)
    max_bytes (word)
    first_disk_buff (dword, ptr)
    curr_dir (dword, ptr)
    lastdrive (byte,hex)
    string_area (dword, ptr)
    size_string_area (word)
    fcb_tbl (dword, ptr)
    fcb_y (word)
    nul (byte,dump,18)

structure list_31 fields            ; DOS 3.1+
    share_retry_count (word)
    retry_delay (word)
    curr_disk_buff (dword, ptr)
    unread_con (word)
    mcb (word)
    dpb (dword, ptr)
    file_tbl (dword, ptr)
    clock (dword, ptr)
    con (dword, ptr)
    max_bytes (word)
    disk_buff (dword, ptr)
    curr_dir (dword, ptr)
    fcb (dword, ptr)
    num_prot_fcb (word)
    num_blk_dev (byte,hex)
    lastdrive (byte,hex)
    nul (byte,dump,18)
    num_join (word)

intercept 21h
    function 52h
        on_exit
            if (OS_MAJOR == 2)
```

```
            output (es:bx[-12]->list_20)
        if (OS_MAJOR == 3) and (OS_MINOR == 0)
            output (es:bx[-12]->list_30)
        if (OS_MAJOR == 3) and (OS_MINOR != 0)
            output (es:bx[-12]->list_31)
            output (es:bx[-12]->list_31.con->byte,dump,18)
            output (es:bx[-12]->list_31.clock->byte,dump,18)
        ; INTRSPY doesn't have >= tests yet
        if (OS_MAJOR == 4) or (OS_MAJOR == 5)
            output (es:bx[-12]->list_31)
    on_exit
        output ""
```

Note that you can display an entire structure with one OUTPUT statement: CMDSPY takes care of formatting output according to the format options specified in the structure itself. When you are displaying an entire structure with a single OUTPUT statement, CMDSPY also takes care of displaying the field names. For example, here is sample output from LSTOFLST.SCR:

```
LIST_31.SHARE_RETRY_      : 0003h
LIST_31.RETRY_DELAY       : 0001h
LIST_31.CURR_DISK_BU      : 09CC:0000
LIST_31.UNREAD_CON        : 0000h
LIST_31.MCB               : 09F3h
LIST_31.DPB               : 028E:75F0
LIST_31.FILE_TBL          : 028E:0098
LIST_31.CLOCK             : 0070:01A4
LIST_31.CON               : 0070:016E
LIST_31.MAX_BYTES         : 0200h
LIST_31.DISK_BUFF         : BC90:0000
LIST_31.CURR_DIR          : 0ACE:0000
LIST_31.FCB               : 0AC0:0000
LIST_31.NUM_PROT_FCB      : 0000h
LIST_31.NUM_BLK_DEV       : 03h
LIST_31.LASTDRIVE         : 05h
LIST_31.NUL               :
 | 3F 00 F4 09 04 80 99 15 9F 15 4E 55 4C 20 20 20 |  |?.........NUL   |
 | 20 20                                            |  |                |
LIST_31.NUM_JOIN          : 9000h

 | 80 01 70 00 13 80 0C 06 17 06 43 4F 4E 20 20 20 |  |..p.......CON   |
 | 20 20                                            |  |                |

 | B6 01 70 00 08 80 0C 06 60 06 43 4C 4F 43 4B 24 |  |..p.....'.CLOCK$|
 | 20 20                                            |  |                |
```

Of course, some other program that actually calls INT 21h Function 52h is needed as a trigger for LSTOFLST.SCR.

Log Your Machine's Activity

Out of UNDOC.SCR comes a nearly ready made mini-application for INTRSPY. Using one section of UNDOC.SCR as a starting point, you can write a script (EXEC.SCR) that will maintain a log of all the programs run on a computer, together with their command-line arguments and their completion codes. This script is shown below:

```
; EXEC.SCR
structure param_block fields
    env_seg (word,hex)
    args (dword,ptr)

intercept 21h
    function 0ah
        on_exit output (ds:dx[1]->byte,string,60)
    function 4bh
        on_entry
            output
                (0:046Ch->dword,dec) "                    ; time
                (ds:dx->byte,asciiz,64)                   ; executable
                (es:bx->param_block.args->byte,string,32) ; cmdline args
        on_exit
            if (cflag == 1) sameline " [FAIL " ax "]"
    function 00h 4ch
        on_entry sameline " [RET " al "]"
    function 31h
        on_entry sameline " [TSR " al "]"

intercept 20h
    on_entry sameline " [RET20 " al "]"

intercept 27h
    on_entry sameline " [TSR27 " al "]"
```

This script reports on all calls to the DOS EXEC function and on all calls to the surprisingly large number of DOS functions and interrupts that handle program termination. INT 21h Function 0Ah (Buffered Keyboard Input) is intercepted so that you can grab the actual command line typed by a user (though note that Function 0Ah can certainly be used by programs other than COM-

MAND.COM). EXEC.SCR also displays the ROM BIOS timer tick, giving a crude display of how much time was spent in each program. Here is sample output from EXEC.SCR after compiling HELLO.C again:

```
cl hello.c
 534139   C:\MSC\BIN\CL.EXE hello.c
 534150   C:\MSC\BIN\c1.exe [RET 00]
 534194   C:\MSC\BIN\c2.exe [RET 00]
 534263   C:\MSC\BIN\c3.exe [RET 00]
 534312   C:\BIN\link.exe @"\"c:\tmp\004951lk\" [RET 00] [RET 00]
cmdspy report
 536537   C:\UNDOC\MAXEY\CMDSPY.EXE report [RET 00]
```

To find the number of seconds spent in a program, subtract its tick count (for example, 534,263) from the tick count for the start of the next program; then divide the result by 18.2 (the number of timer ticks per second).

EXEC.SCR points up an interesting "wish list" item for INTRSPY. Notice that all the termination functions use SAMELINE so that the program termination status will be printed on the same line as the invocation. If, however, you run a program that spawns a subprogram, as CL.EXE does, the return code for the top level program will not appear on the appropriate line. A future version of INTRSPY should auto-indent, in which case the EXEC.SCR code above would have the SAMELINEs replaced by OUTPUTs. Even better, INTRSPY should allow an output stack, so that results could be logically linked. But as it is, EXEC.SCR accomplishes a useful function for which others have written complete utilities. The above script took a few minutes to code, test, and refine.

Monitoring Disk I/O

The next application took longer to develop, but after you have read it, imagine how long it would take to write from scratch.

The idea behind DISK.SCR is to log DOS file system calls made by specific programs, as well as the related BIOS disk activity generated by such calls. Note that this script uses the RUN command to run a specific program that you want to monitor. This helps you watch, as an example, FORMAT.COM, without inadvertently also watching COMMAND.COM:

```
;   DISK.SCR
;   This script relates DOS disk calls to the hard disk
;   BIOS calls involved.
```

```
;

; DOS 4+/Compaq DOS 3.31+ >32M partition
structure big fields
    sector (dword,hex)
    num (word,hex)
    addr (dword,ptr)

intercept 21h
    function 32h
        on_entry output "2132: Get DPB drive " dl
    function 40h
        ; hook Write just to see messages (e.g., from FORMAT)
        on_entry if (bx == 1)   ; stdout
            output (ds:dx->byte,asciiz,cx)
    function 44h
        subfunction 09h
            on_entry output "214409: IOCTL drive " bl " Remote? "
        subfunction 0dh
            on_entry output "21440D: IOCTL drive " bl
                if (cl == 40h) sameline " [40: Set Device Parameters]"
                if (cl == 42h) sameline " [42: Format and Verify Track]"
                if (cl == 60h) sameline " [60: Get Device Parameters]"
        subfunction 0fh
            on_entry output "21440F: IOCTL Set Logical Drive " bl
            on_exit if (cflag == 0) sameline " ==> " al
    function 60h
        on_entry output "2160: Canon " (ds:si->byte,asciiz,32) " ==> "
        on_exit sameline (es:di->byte,asciiz,32)

intercept 25h
    on_entry
        output "25: Abs Disk Read drv " al ", at sectr "
        if (cx == 0FFFFh)
            sameline (ds:bx->big.sector) ", "
                     (ds:bx->big.num) " sctrs"
        if (cx != 0FFFFh)
            sameline dx ", " cx " sctrs"
    on_exit if (cflag==1) sameline " [fail]"

intercept 26h
    on_entry
        output "26: Abs Disk Write drv " al ", at sectr "
        if (cx == 0FFFFh)
            sameline (ds:bx->big.sector) ", "
                     (ds:bx->big.num) " sctrs"
        if (cx != 0FFFFh)
```

```
            sameline dx ", " cx " sctrs"
    on_exit if (cflag==1) sameline " [fail]"

intercept 13h
    function 0 on_entry output "1300: Recalibrate drive " dl
    function 1 on_exit output "1301: Disk system status " al
    function 2
        on_entry
            output "1302: Read " al " sctrs: drv " dl ", head " dh
                ", sctr " cl ", trk " ch
        on_exit if (cflag==1)
            sameline " - FAILED (" ah ")"
    function 3
        on_entry
            output "1303: Write " al " sctrs: drv " dl ", head " dh
                ", sctr " cl ", trk " ch
        on_exit if (cflag==1)
            sameline " - FAILED (" ah ")"
    function 4
        on_entry
            output "1304: Verify " al " sctrs: drv " dl ", head " dh
                ", sctr " cl ", trk " ch
        on_exit if (cflag==1)
            sameline " - FAILED (" ah ")"
    function 5
        on_entry
            output "1305: Format " al " sctrs: drv " dl ", head " dh
                ", sctr " cl ", trk " ch
        on_exit if (cflag==1)
            sameline " - FAILED (" ah ")"
    function 8
        on_entry
            output "1308: Get drive params for " dl
        on_exit
            if (cflag==1) sameline " - FAILED (" ah ")"
            if (cflag==0)
                output "Type " bl ", " dl " drvs, max head " dh
                    ", max sctr " cl ", max cyls " ch
    function 0ch
        on_entry output "130C: Seek to cyl " ch ", drv " dl ", head " dh
        on_exit if (cflag==1) sameline " - FAILED (" ah ")"
    function 0dh
        on_entry output "130D: Alternate reset drive " dl
        on_exit if (cflag==1) sameline " - FAILED (" ah ")"
    function 10h
        on_entry output "1310: Test drive " dl
        on_exit sameline " - status " ah
```

```
    function 15h
        on_entry output "1315: Get type drv " dl
        on_exit
            sameline ": "
            if (ah==0) sameline "No disk present"
            if (ah==1) sameline "Floppy - Not changed"
            if (ah==2) sameline "Floppy - changed"
            if (ah==3) sameline "Fixed disk"
        on_exit
            sameline " : sctrs " cx dx
    function 16h
        on_entry output "1316: Get media change drv " dl ": "
        on_exit
            if (ah==0) sameline "Unchanged"
            if (ah==6) sameline "Changed"
    function 17h
        on_entry
            output "1317: Set type drv " dl ": "
            if (al==0) sameline "no disk"
            if (al==1) sameline "reg disk in reg drv"
            if (al==2) sameline "reg disk in high dens. drv"
            if (al==3) sameline "high dens. disk in high dens. drv"
            if (al==4) sameline "720k disk in 720k drv"
            if (al==5) sameline "720k disk in 1.44M drv"
            if (al==6) sameline "1.44M disk in 1.44M drv"
    function 18h
        on_entry
            output "1318: Set media type drv " dl ": sctrs/trk " cl
                ", trks " ch ": "
        on_exit
            if (ah==0) sameline "OK"
            if (ah==1) sameline "Not available"
            if (ah==0ch) sameline "Not supported"
            if (ah==80h) sameline "No disk in drive"

run "%1 %2 %3 %4"
report "disk.out"
report ""
stop
```

Note that the script reports once to a file and once to the screen. It is sometimes useful to see the output immediately, and then to be able to review it in an editor. Below is the invocation and the output it generated while formatting a double-density 5.25" diskette on a Compaq 386 system. You can see that there is also a small amount of output not generated by FORMAT itself: the first four

lines of the CMDSPY REPORT also show DOS loading FORMAT.COM from the hard disk (drive 80h):

```
C:\>intrspy -r10480 -i4096
INTRSPY v1.00 is now resident.

C:\>cmdspy compile disk format a: /4
Compiling DISK.SCR
Generated  231 bytes of code for interrupt 21h.
Generated   53 bytes of code for interrupt 25h.
Generated   53 bytes of code for interrupt 26h.
Generated  940 bytes of code for interrupt 13h.
Running C:\DOS33\FORMAT.COM
Insert new diskette for drive A:
and strike ENTER when ready

Format complete

    362496 bytes total disk space
    362496 bytes available on disk

C:\DOS\FORMAT.COM terminated (0)
INTRSPY returned a total of 10032 bytes of results.

1302: Read 01 sctrs: drv 80, head 01, sctr 0A, trk 12
1302: Read 01 sctrs: drv 80, head 01, sctr 03, trk 00
1302: Read 1A sctrs: drv 80, head 01, sctr 0A, trk 12
1302: Read 01 sctrs: drv 80, head 03, sctr 02, trk 12
214409: IOCTL drive 01 Remote?
2160: Canon  A:CON ==>  A:/CON
21440D: IOCTL drive 01 [60: Get Device Parameters]
21440D: IOCTL drive 01 [40: Set Device Parameters]
21440F: IOCTL Set Logical Drive 01 ==> 01
 Insert new diskette
 for drive A:
and strike ENTER when
ready
 21440D: IOCTL drive 01 [42: Format and Verify Track]
1318: Set media type drv 00: sctrs/trk 09, trks 27: Not available
 Head:   0 Cylinder:
    0
21440D: IOCTL drive 01 [42: Format and Verify Track]
1318: Set media type drv 00: sctrs/trk 09, trks 27: Not available
1317: Set type drv 00: reg disk in high dens. drv
1305: Format 09 sctrs: drv 00, head 00, sctr 00, trk 00
1304: Verify 09 sctrs: drv 00, head 00, sctr 01, trk 00
```

```
 Head:   1 Cylinder:
    0
21440D: IOCTL drive 01 [42: Format and Verify Track]
1318: Set media type drv 00: sctrs/trk 09, trks 27: Not available
1305: Format 09 sctrs: drv 00, head 01, sctr 00, trk 00
1304: Verify 09 sctrs: drv 00, head 01, sctr 01, trk 00
 Head:   0 Cylinder:
    1
21440D: IOCTL drive 01 [42: Format and Verify Track]
1318: Set media type drv 00: sctrs/trk 09, trks 27: Not available
1305: Format 09 sctrs: drv 00, head 00, sctr 00, trk 01
1304: Verify 09 sctrs: drv 00, head 00, sctr 01, trk 01
 Head:   1 Cylinder:
    1
21440D: IOCTL drive 01 [42: Format and Verify Track]
1318: Set media type drv 00: sctrs/trk 09, trks 27: Not available
1305: Format 09 sctrs: drv 00, head 01, sctr 00, trk 01
1304: Verify 09 sctrs: drv 00, head 01, sctr 01, trk 01
    .
    .
    [ same sequence of 21440D/1318/1305/1304 calls for
      trk 02/head 00 through trk 26/head 01 ]
    .
    .
 Head:   1 Cylinder:
   39
21440D: IOCTL drive 01 [42: Format and Verify Track]
1318: Set media type drv 00: sctrs/trk 09, trks 27: Not available
1305: Format 09 sctrs: drv 00, head 01, sctr 00, trk 27
1304: Verify 09 sctrs: drv 00, head 01, sctr 01, trk 27
 Format complete
26: Abs Disk Write drv 00, at sectr  00000000h,  0001h sctrs
1303: Write 01 sctrs: drv 00, head 00, sctr 01, trk 00
26: Abs Disk Write drv 00, at sectr  00000001h,  0002h sctrs
1303: Write 02 sctrs: drv 00, head 00, sctr 02, trk 00
26: Abs Disk Write drv 00, at sectr  00000003h,  0002h sctrs
1303: Write 02 sctrs: drv 00, head 00, sctr 04, trk 00
26: Abs Disk Write drv 00, at sectr  00000005h,  0001h sctrs
1303: Write 01 sctrs: drv 00, head 00, sctr 06, trk 00
26: Abs Disk Write drv 00, at sectr  00000006h,  0001h sctrs
1303: Write 01 sctrs: drv 00, head 00, sctr 07, trk 00
26: Abs Disk Write drv 00, at sectr  00000007h,  0001h sctrs
1303: Write 01 sctrs: drv 00, head 00, sctr 08, trk 00
26: Abs Disk Write drv 00, at sectr  00000008h,  0001h sctrs
1303: Write 01 sctrs: drv 00, head 00, sctr 09, trk 00
26: Abs Disk Write drv 00, at sectr  00000009h,  0001h sctrs
1303: Write 01 sctrs: drv 00, head 01, sctr 01, trk 00
```

```
26: Abs Disk Write drv 00, at sectr  0000000Ah,  0001h sctrs
1303: Write 01 sctrs: drv 00, head 01, sctr 02, trk 00
26: Abs Disk Write drv 00, at sectr  0000000Bh,  0001h sctrs
1303: Write 01 sctrs: drv 00, head 01, sctr 03, trk 00
21440D: IOCTL drive 01 [40: Set Device Parameters]
2132: Get DPB drive 01
1302: Read 01 sctrs: drv 00, head 00, sctr 01, trk 00
1302: Read 01 sctrs: drv 00, head 00, sctr 06, trk 00
1302: Read 01 sctrs: drv 00, head 00, sctr 02, trk 00
1302: Read 01 sctrs: drv 00, head 00, sctr 01, trk 00
1302: Read 01 sctrs: drv 00, head 00, sctr 06, trk 00
     362496 bytes total
disk space
     362496 bytes available
on disk
 Format another (Y/N)
 ?
21440D: IOCTL drive 01 [40: Set Device Parameters]
```

Whereas DISK.SCR groups functions together by interrupt and function number, the output resulting from running the script together with the DOS FORMAT command is quite different. Here you see the nesting of BIOS calls from DOS calls. In the above display, you can clearly see how FORMAT displays the "Head: XX Cylinder: XX" odometer (which we're able to show here by monitoring INT 21h Function 40h). It then calls DOS INT 21h Function 44h Subfunction 0Dh to format and verify a track, which, in turn, calls BIOS INT 13h Functions 18h, 05h, and 04h to format and verify the sectors that make up the track. (INT 13h Function 18h returns "Not available" here because that function is not provided by the Compaq ROM BIOS.)

After the format itself is complete, as the above INTRSPY output shows, to create the FAT and root directory on the newly formatted disk, FORMAT calls the DOS Absolute Disk Write interrupt (INT 26h). This interrupt, in turn, calls the BIOS Write Sector Function (INT 13h Function 03h). Note also how DISK.SCR handles INT 25h and INT 26h calls on systems with partitions larger than 32 megabytes. This is important even when you are formatting a floppy disk, because even then FORMAT will use the alternate form of INT 26h, where CX holds the value FFFFh and DS:BX points to a structure that in DISK.SCR is called BIG.

You can also see from this display that FORMAT uses two undocumented DOS calls: Resolve Path String to Canonical Path String (INT 21h Function 60h)

and Get DPB (INT 21h Function 32h). Again, having INTRSPY means that you can do this kind of exploring without disassembling.

MEM

One last INTRSPY script worth examining is MEM.SCR, which, in 24 lines of INTRSPY code, can monitor all DOS memory allocation by intercepting INT 21h Functions 48h (Allocate), 49h (Release), and 4Ah (Resize). Again, Function 4Bh (EXEC) is monitored as well, so that you know which program performed the memory operation:

```
; MEM.SCR
intercept 21h
    function 48h
        on_entry
            output "ALLOC " bx "h paras"
        on_exit
            if (cflag==1)
                sameline " FAIL (" ax "), only " bx "h available"
            if (cflag==0)
                sameline " - seg " ax "h"
    function 49h
        on_entry output "FREE seg " es "h"
        on_exit if (cflag==1) sameline " denied (" ax "h)"
    function 4ah
        on_entry
            output "REALLOC seg " es "h to " bx "h paras"
        on_exit
            if (cflag==1)
                sameline " FAIL (" ax "h), only " bx "h available"
    function 4bh
        on_entry output (ds:dx->byte,asciiz,64)
```

MEM is used in tracking down memory allocation bugs, but it is also useful in a hands-on examination of the DOS memory allocation issues discussed by in chapter 3 of this book. For example, you can see immediately that DOS programs typically are allocated all available memory:

```
C:\UNDOC\MAXEY\HELLO.EXE
ALLOC FFFFh paras FAIL (0008), only A1E5h available
ALLOC A1E5h paras - seg 161Bh
FREE seg 161Bh
```

Space does not allow me to show all the uses that even my limited imagination has found for INTRSPY. Apart from the mini-applications shown above, there are many scripts included on the accompanying disk that you might find instructive and that should in any case help you in the implementation of your own ideas.

Writing a Generic Interrupt Handler

Let's now step back and discuss some of the design issues behind INTRSPY. Some of this sounds like a functional specification for INTRSPY, because, in fact, it comes from the functional specification we drew up for INTRSPY.

Traditionally, DOS programmers would write a host of small, tailor-made programs to perform the kind of exploration we have been doing. In the past, I have written *separate* programs to monitor NetBIOS (INT 5Ch) calls, DOS (INT 21h) calls, and EMS (INT 67h) calls. The cycle is to write a simple TSR, debug it, and eventually have something that could printf or writeln register values into or out of intercepted functions. In principle, that initial version cost a few hours of thought, programming, and debugging effort. Subsequent versions that refined the list of functions being monitored or added a new or increased capability came easier and more quickly, but they still required recompilation and debugging.

All monitoring programs are essentially the same, however, whether it is DOS, NetBIOS, EMS, or anything else that is being monitored. This suggests that it should be possible to write a *generic* monitoring program. By extension, this means writing a generic interrupt handler. The actual interrupts, functions, and/or subfunctions you wished to monitor would be *parameters* to the generic interrupt handler.

Rapid modification of the parameters, without the necessity of debugging, is a high priority to ensure that the building of the tools doesn't detract, or distract, from your investigations. The sheer number of parameters that need to be within reach, however, and the need for a broad range of capabilities, suggest that a command-line switch based TSR is unlikely to be adequate. A script interpreting, precompiled interrupt handling tool would be better.

Because DOS uses a number of different interrupt numbers and provides its services through functions and subfunctions within these interrupts (generally specified via AH and AL), and because you would want to be able to monitor these interrupts and their function calls selectively, your tool must be capable of monitoring any subfunction of any function of any interrupt. It should, therefore,

allow you to build some useful logging and exploration applications, not just in the field of undocumented DOS calls but in a whole range of DOS-related and non-DOS-related interrupt-based services. Logging DOS memory usage would be a snap, for example. Relating DOS disk access to the underlying BIOS interrupts, watching EMS calls, recording a NetBIOS session—all these should be within reach simply by using different, easily modifiable parameters: snap-on debug tools for PC software developers!

The preliminary specification is beginning to form:

The program should be script-driven. This allows for transportable, repeatable, and canned experiments and debugging tools.

The program should intrude as little as possible—that is, it should make no DOS calls, should hook few interrupts, and should consume little memory. Any interrupts that are generated, any memory that is consumed, any interrupts that are intercepted other those that are supposed to be intercepted—all these factors constitute noise and can affect, potentially in many ways, the systems being studied. If CMDSPY made undocumented DOS calls, for example, these would show up in the report from UNDOC.SCR! Thus, it is perhaps ironic that in a book devoted to the advertisement of undocumented DOS calls as crucial tools in the development of TSRs and system level software, the source for CMDSPY contains no undocumented DOS calls (INTRSPY.EXE does, however). Once it is resident, INTRSPY generates no interrupts and uses no DOS services whatsoever.

The program should provide dumps of register contents, flags, and DOS and other structures in memory. The STRUCTURE capability was specified initially because of the DOS List of Lists structure and the associated DOS Function 52h (see LSTOFLST.SCR above). As was hinted at above, however, once an initial capability of this type has been provided, building in a little flexibility can make it useful in many originally unimagined ways.

As was already said, the central requirement is to be able to focus, from session to session, on different subfunctions of different functions of different interrupts, and reconfiguration should be easy and painless so that it does not impede learning, experimentation, or the debugging progress.

Access should be available both before and after an interrupt is serviced. The program needs to be able to see and act on the parameters that the caller supplies, and the structures, registers, and flags that the interrupt function returns.

A non-resident transient portion should compile the scripts, hand them off to the resident portion, and decode the output. When we were first discussing the

specification for INTRSPY, we had been using some of the other available interrupt monitoring software packages, and it was not clear that any of them offered the flexibility we needed. The options here were either a shell implementation or a TSR/transient controller combination.

In a shell implementation, programs under scrutiny would be run from within the system. Much like those of a debugger, a shell implementation's facilities would be available until the user quit the user interface. The problems with this approach include potentially much reduced memory availability for the program being studied and less flexibility to run commands and batch files as well as programs. Benefits include control over the environment and the ability to limit monitoring to a specific run of a particular program. Thus, INTRSPY includes this as an *option* with the RUN statement.

We decided to go with the TSR/transient controller combination. A relatively small TSR would perform the monitoring function, and a separate, less memory-constrained program would perform script compilation, result formatting and printing, and would handle all communications with the TSR. We saw this approach as preferable, because the transient portion itself can be made to act as a shell. This approach does have its disadvantages, however. Complexity is introduced through the decoupling of the compilation from the execution of the script. This is awkward only because, unlike in normal compilation and execution cycles, script source is needed to be able to decode the results of the execution.

Now, let us impose two constraints on the system:

The resident portion will store results, rather than write them to disk or pop-up screens. This will remove the need to plan for file I/O within the generic interrupt service routine (ISR) code. In addition, it keeps us closer to our second specification objective. File I/O by necessity generates interrupts, changes the state of the operating system by a little or a lot, and leads to coding complexity when implemented properly in TSRs (as was shown in chapter 5).

The script language need not provide complex conditional test capabilities. Equality/inequality is an adequate test, and left-to-right non-nested test and/or sub-tests will also provide enough conditional power. Initially, registers and flags are sufficient as targets. Adding other targets would require changes to the script language compiler in CMDSPY, but would involve little or no change to the conditional test structures nor to the code in INTRSPY.

The Problem with Intel's INT

Now that we have explained how to use INTRSPY and discussed some of the thinking behind it, we need to discuss briefly some problems with the Intel INT instruction. These problems stand in the way of anyone seeking to write a generic interrupt handler. The INT instruction is the source of a great deal of the flexibility in the PC architecture, because the ability to get and set interrupt vectors means that system services (including DOS itself) are infinitely extensible, replaceable, and monitorable. Yet, given its importance, the INT instruction is also remarkably inflexible in two key ways:

- An interrupt handler does not know which interrupt number invoked it.
- The INT instruction itself expects an *immediate* operand: you cannot write MOV AX, 21h, and then INT AX; you must write INT 21h.

The first problem raises the question, How will the program trap a variable number of user-specifiable interrupts? There are at least three possibilities here.

The first possibility is to use a generic interrupt service routine (ISR) for all interrupts the user specifies. When invoked, the ISR can use the return address on the stack to find out what INT instruction issued the interrupt. Here is an example of a Turbo Pascal interrupt procedure that would attempt to find out what interrupt it was hanging off:

```
procedure GenericISR(flags,ip,cs,ax,bx,cx,dx,si,di,ds,es,bp:word);
var
    MyInterruptNum : byte;
begin
    MyInterruptNum:=byte(ptr(cs,pred(ip))^);
        .
        .
        .
end;
```

Simple, isn't it? This would indeed be an elegant, economical solution. Unfortunately, it is an unreliable strategy because many high-level language compilers compile interrupts into PUSHF and far CALL instruction sequences, rather than do an actual INT. Other compilers push the address of the handler on the stack and do RETF to it. In order to cover the different possibilities to simulate an INT instruction, a generic ISR would need a small disassembler. The reason for all these different ways of performing what should simply be an INT is in fact the

second problem noted above: that the INT instruction itself can't be parameterized. Thus, different compiler vendors implement functions such as int86x() differently.

The other crippling failure of the above code is that it relies on the caller's stack being large enough for our interrupt processing. Bear in mind that the program is going to have to be able to handle DOS internal function calls, for which DOS will have switched to its own small stack (less than 400 bytes; for exact sizes, see the appendix entry for INT 21h Function 5D06h). In the above example, 19 bytes of stack have been used by the Turbo Pascal compiler to save registers and for the local variable before we call our first procedure or use any more local variables. That is unacceptable.

Another possibility involves coding 256 small stubs. Only those interrupts the user specifies would be redirected to the appropriate stubs. When invoked, a stub would record its interrupt number, save away the caller's stack, switch to the program's internal stack, then call the generic ISR. This is a better solution but will waste some kilobytes of memory in implementation code.

There are variations on the above, but none were appealing, and I decided to go with what I think is at least a more interesting solution—that is to allocate a custom ISR "object" on the heap for each interrupt to be monitored. I call it an *object* because it is a structure containing machine code to perform stack switching, registers for saving state, and the specific "compiled" code associated with the interrupt processing. INTRSPY contains a skeleton ISR that is copied onto the heap and "fixed up" with some run-time dependent addresses. The interrupt processing is then actually performed by procedures called from the ISR machine code on the heap.

Will the program have to fix up the relative data structure offsets at run time, because it is allocating on the heap and therefore cannot expect every ISR structure to be paragraph-aligned? No: each ISR starts on a paragraph boundary, wasting up to 15 bytes to ensure it. This is an insignificant amount and allows the entrypoint, as well as all the data fields in the ISR structure, to be at a constant offset. This is, after all, what DOS does. INTRSPY can thus be viewed as a program loader for which the programs are just small pieces of ISR code and the handler space substitutes for DOS memory.

How should the interrupt processing instructions be stored? My first reaction was to think in terms of compilation into machine language. Instead, I implemented a linked tree of small structures that describes the entities in the process-

ing. The resident ISR processor walks the linked list of function records stored for the interrupt that has been intercepted, processing the subfunction branches of each. Each subfunction branch is a linked list of subfunction records, each of which points to a "before" and "after" branch. These branches are, in turn, linked lists of records that specify the conditionals and resulting data storage to be performed before and after the interrupt is serviced.

Implementation

Let us take a look at the structure of INTRSPY and its transient controller CMDSPY. The system functions as follows:

- INTRSPY is loaded and reserves some memory for holding interrupt processing instruction records and results data.
- CMDSPY compiles the specified script file into a tree structure of records on the basis of STRUCTURE and INTERCEPT constructs. The compiler is a fairly rudimentary finite state machine that implements the language as it is specified in the earlier section called "INTRSPY System User Guide." CMDSPY passes the compiled INTERCEPT code interrupt number by interrupt number to INTRSPY, which generates a new ISR record on the heap for it and copies in the instruction records.

When all the interrupts have had their instructions loaded, CMDSPY passes the name of the input file that has just been processed, the drive and directory that were current when it was processed, and any command-line arguments that were passed to the script to INTRSPY for later retrieval.

Next, CMDSPY tells INTRSPY to start watching the interrupts and performing the instructions that it has just been passed. Up until this point, the ISR records have been on the heap, but their addresses have not been set into the interrupt vector table. The START command makes the vectors of all the intercepted interrupts point to the appropriate heap ISR stub entry point.

Programs are run, commands are issued, and INTRSPY builds up the results in memory. When results are to be output, CMDSPY sends a command to the INTRSPY to stop monitoring. INTRSPY restores the vectors of the intercepted interrupts to what they were before the above start command—that is, it switches itself out of the interrupt chains (if it can).

CMDSPY allocates a buffer and requests INTRSPY to download the results, which take the form of a linked list of variable length output records.

Although the script language supports structure definitions and literal strings for output, these compile into offsets, lengths, and reference numbers. In order to decode the results that are returned from INTRSPY, CMDSPY must recompile the script. In order to do that relatively safely, it must make current the drive and directory that were current at the time the script was first compiled, load the file as it was specified on the command line the first time, and pass to it the same command-line arguments as it was passed the first time. This is why CMDSPY had the INTRSPY store those pieces of information after loading the interrupt instructions.

Having reloaded the original script file, CMDSPY walks the output chain formatting and printing the data in the record structures.

INTRSPY's Interrupt Processing

The machine language generic ISR stub that will be cloned for each interrupt is stored as an array of bytes, not as a procedure. INTRSPY treats this code not as code at all but as data to be copied, modified at run time, and referenced as an element in a record structure. Only other programs treat this object as code: these other programs will (unwittingly) invoke it by issuing INT instructions, but INTRSPY itself doesn't. So the ISR stub is implemented to support that usage:

```
const
    isr_code_max = 115;      { offset of last byte of code in isr }

type
    isr_code_buffer = array[0..isr_code_max] of byte;
                                        { our heap based interrupt  }
                                        { service routine is 116    }
                                        { bytes long                }

{ Skeleton generic interrupt handler code - needs "fixing up" before    }
{ being cloned onto on the heap.                                        }
interrupt_code : isr_code_buffer =
    (       $90,                  { nop OR int 3          ; for debugging }
    $2e,$8f,$06, save_ip_ofs, 0,{ pop   cs:save_ip      ; store cs:ip   }
    $2e,$8f,$06, save_cs_ofs, 0,{ pop   cs:save_cs                      }
    $2e,$8f,$06, save_fl_ofs, 0,{ pop   cs:save_fl      ; and flags     }

    $2e,$89,$26, save_sp_ofs, 0,{ mov   cs:save_sp,sp   ; save stack    }
        $8c,$d4,                  { mov   sp,ss                         }
    $2e,$89,$26, save_ss_ofs, 0,{ mov   cs:save_ss,sp                   }
```

```
    $bc,     0,0,           { mov   sp,SSEG          ; set our stack }
    $8e,$d4,                { mov   ss,sp                           }
    $bc,     0,0,           { mov   sp,SPTR                         }

    $9c,                    { pushf                  ; call our pre- }
    $9a,     0,0,0,0,       { call  prepost          ; intr proc.    }

$2e,$8b,$26, save_ss_ofs, 0,{ mov   sp,cs:save_ss    ; put back      }
    $8e,$d4,                { mov   ss,sp            ; caller's stack}
$2e,$8b,$26, save_sp_ofs, 0,{ mov   sp,cs:save_sp                    }

    $9c,                    { pushf                  ; caller's flags}
    $9a,     0,0,0,0,       { call  old_int          ; gen interrupt }

    $9c,                    { pushf                                 }
$2e,$8f,$06, save_fl_ofs, 0,{ pop   cs:save_fl       ; save ret flags}

$2e,$89,$26, save_sp_ofs, 0,{ mov   cs:save_sp,sp    ; save stack    }
    $8c,$d4,                { mov   sp,ss                           }
$2e,$89,$26, save_ss_ofs, 0,{ mov   cs:save_ss,sp                    }

    $bc,     0,0,           { mov   sp,SSEG          ; set our stack }
    $8e,$d4,                { mov   ss,sp                           }
    $bc,     0,0,           { mov   sp,SPTR                         }

    $9c,                    { pushf                  ; call our post }
    $9a,     0,0,0,0,       { call  prepost          ; intr proc.    }

$2e,$8b,$26, save_ss_ofs, 0,{ mov   sp,cs:save_ss    ; put back      }
    $8e,$d4,                { mov   ss,sp            ; caller's stack}
$2e,$8b,$26, save_sp_ofs, 0,{ mov   sp,cs:save_sp                    }
$2e,$ff,$36, save_cs_ofs, 0,{ push  cs:save_cs       ; restore       }
$2e,$ff,$36, save_ip_ofs, 0,{ push  cs:save_ip       ; return addr.  }
    $fb,                    { sti                    ; enable intrs  }
    $cb                     { ret                    ; ret NOT iret  }
    );
```

The interrupt processing instruction records take the form of a linked tree of variable length records. These records are linked not through normal Pascal 4-byte (far) pointers, but by 2-byte offset-to-next fields to save space. They are processed by pre_post, which is called directly from the ISR stub. The pre_post procedure is implemented as a Turbo Pascal interrupt type procedure for two reasons. First, the program needs access to all the registers on entry, and an interrupt type procedure provides access to most of them; and second, the program

needs DS set up, and an interrupt procedure takes care of that as well. The registers that are needed and that the Pascal interrupt procedure cannot make available are the CS, IP, SS, SP, and FLAGS as they were on entry to the ISR. These are saved by the ISR stub in the fields described above. The processing performed by pre_post and its subordinate procedures is relatively simple, and a high-level pseudocode description looks something like this:

```
If (this is pre-interrupt processing) then
    if this is DOS Int 21h, grab the Ctrl-C and CritErr vectors
    if (this is a DOS EXEC i.e. Int 21h/AH=4Bh) and
        (a Ctrl-C or Critical Err has resulted in termination) then
            clear down the function/subfunction (fxn/sfxn) stack
            to the previous DOS EXEC or start-of-stack, whichever
            comes first
    push AX (function and subfunction) on the fxn/sfxn stack
otherwise, since it is post-interrupt processing,
    set function and subfunction from AX popped off the fxn/sfxn stack

If (we want this function) or (we want all functions)
    then
        if (we want this subfunction) or (we want all subfunctions)
            repeat
                if we have a real (non-dummy) conditional to perform,
                    repeat
                        evaluate a test
                        go on to the next test
                            and apply the last 'AND' or 'OR'...
                     until one fails, or all done, and success
                if success, or we had a dummy conditional
                    then
                        repeat
                            process an output request
                            go onto the next output request
                         until all done
                go on to the next conditional
               until there are no more real or dummy conditionals

If (this is post-interrupt processing) or
    (this is a DOS termination function i.e.
        ints 20h, 27h; int 21h fxns 0, 31h 4ch) then
    pop AX off the fxn/sfxn stack
    if (we have been asked to 'switch off' but were in mid-EXEC) then
        attempt to switch off
```

Three related complications in the above logic must be explained. They are there to cope with a special type of interrupt and function—namely, the DOS termination functions. This is because these functions, when invoked, do not return. I only realized this fairly obvious fact during a debugging session that had dragged on into the night while INTRSPY was being developed. After some time with a script running, INTRSPY would just lock up my machine. I traced the problem fairly rapidly to a corrupted register save field in the ISR record in the handler space. I had allowed a function/subfunction stack of 16 entries, and it became clear that the stack was being overflowed and was spilling out into the rest of the record. As soon as any of the saved SP, SS, CS, or IP values were corrupted, INTRSPY returned to forever! I knew that DOS did not nest that deep, and then it hit me. . .

Specifically, then, a DOS intercepter must tidy up the stack while still in preprocessing mode, because, unlike "normal" interception, the DOS intercepter will not have an opportunity to do so during post-processing.

The situation is complicated still further by Ctrl-Break and Critical Errors. When a DOS function that checks for these conditions gives control to the DOS break processor, in all cases it aborts the currently executing function and backs up its internal state to the way it was on entry to the DOS function dispatcher. It then either restarts the original function or, in the case of an Abort, replaces the original function request in AH with a termination request. In that case the INTRSPY intercepter does not receive control back from the function it is currently intercepting, and the function stack is now misaligned. The solution is for INTRSPY to intercept INT 23h (Ctrl-C) and 24h (Critical Error) and "recalibrate" the stack on entry to the next function intercepted if the application handler requests termination. This involves decrementing the function stack pointer until the previous EXEC or start-of-stack, whichever comes first.

Another DOS-specific complication that had to be considered involves the "switch INTRSPY off" request sent by CMDSPY, in response to which INTRSPY attempts to restore the interrupt vectors for the interrupts that it is monitoring. If one of those interrupts is 21h, CMDSPY itself is running as part of a DOS EXEC (Function 4Bh), in which case the ISR handling it is in mid-interrupt. Restoring the vector and potentially replacing that ISR with another one would be disastrous—the DOS EXEC function would return to an address that had been superseded. So the interrupt vector is not restored and the ISR is not marked overwriteable unless the fxn/sfxn stack pointer in the ISR record is 0. Obviously,

in the case of an INT 21h ISR, the fxn/sfxn stack pointer can never be 0 when the "stop monitoring" command is received. The answer is to set a flag in the ISR record that says "when you have finished exit processing and the fxn/sfxn stack pointer is 0, stop monitoring."

The key thing to remember here is that all this complication is encapsulated within the generic interrupt handler in INTRSPY. You can write INTRSPY scripts happily without agonizing over the usual complications of interrupt handling, because INTRSPY's job is to handle these details. This is a key benefit to using such an interrupt-handling language.

Appendix A

Undocumented DOS Functions

Acknowledgment

The material in this appendix has been excerpted from a comprehensive list of IBM PC interrupt calls available for free (and updated several times per year) on bulletin board systems world-wide as well as CompuServe. At the time of printing, the Interrupt List was over 380 pages in length, and growing at a rate of about eight pages per month. Nearly one hundred people have contributed to this massive listing maintained by Ralf Brown; major contributors for the undocumented INT 21h calls include Robin Walker (DOS 3.0, 3.3, and 4.0), Duncan Murdoch (various undocumented fields and data structures), Wes Cowley (many SHARE hooks), Richard Marks (undocumented FindFirst fields), and Ralf Brown (DOS 2.x, 3.1, and some DOS 3.3 and 4.0). For the INT 2Fh calls, major contributors include Robin Walker (much of the DOS 4.0 information) and Ralf Brown (much of the DOS 3.x information).

Sample Entry

The following sample entry illustrates the various conventions used in this appendix by way of a mythical function call. A detailed explanation of various features follows the sample entry.

INT 2Bh Function 01h DOS 2.7x only

GET DWIM INTERPRETER PARAMETERS

The DWIM (Do-What-I-Mean) interpreter is a loadable module called by COMMAND.COM when it encounters an unknown command or syntax error. The DWIM interpreter attempts to determine what the user meant, and returns that guess to COMMAND.COM for execution.

Call with:

AH	01h
DS:SI	pointer to buffer for parameter table (see below)
CX	size of buffer

Returns:

CF set on error

AX error code

01h buffer too small (less than two bytes)

CF clear if successful

CX size of returned data

DX number of times DWIM interpreter invoked since parameters were last set

Notes:

- The DWIM interpreter was apparently never released due to its large memory requirements and slow operation.
- If the given buffer size is at least two bytes but less than the size actually used by the parameter table, the first CX bytes will be copied into the buffer but no error will be returned.

Format of parameter table:

Offset	Size	Description
00h	WORD	size in bytes of following data
02h	WORD	maximum number of error corrections per line to attempt
04h	WORD	order in which to correct errors 00h left-to-right 01h right-to-left *02h most serious first*

DOS 2.70

06h	*BYTE*	*unknown*
07h	DWORD	pointer to DWIM statistics record (see *INT 2B/AH=03h*)

DOS 2.71

06h	*WORD*	*unknown*
08h	DWORD	pointer to DWIM statistics record (see *INT 2B/AH=03h*)

Note:
The statistics record may be placed in a user buffer by changing the pointer with INT 2B/AH=02h.

See Also: *INT 21/AH=0Ah*, INT 2Bh/AH=02h, INT 2Fh/AX=AE01h

The right-hand side of the header indicates the versions of DOS for which the function is valid (in this case, versions 2.70 and 2.71 only). Two other variations are also used: DOS 2+ indicates that the entry is valid for all known versions of DOS from 2.00 to 4.0x inclusive, and DOS 3.1-3.3 indicates that the entry is valid for DOS versions 3.1 through 3.3, inclusive.

Various interrupt functions are called internally by DOS, and should be implemented by a user program rather than called. To further distinguish these functions, "Called with" is used instead of "Call with." For example, INT 2F/AH=11h.

"Pointer to" means that the register or register pair contains the address of the indicated item, rather than the item itself.

Register descriptions which are indented mean that the register only applies for the value indicated by the preceding unindented register description. For this example, the value in AX is meaningful only if the carry flag is set on return, while CX and DX are only meaningful if the carry flag is clear.

Italicized text indicates that the information is not entirely certain, and that particular care (even more than usual for undocumented information) should be exercised when attempting to use it.

A Note: or Notes: section may apply either to the function in general or to the description of a data structure. Notes which come immediately after a Return section apply to the function in general, while those which follow a data structure description apply only to the data structure they follow.

Many data structures change their layouts between versions of DOS. To save space, the different versions are usually merged into a single description. The fields are assumed to be common to all versions unless a version indicator precedes them. In this example, the first

three fields are common to both 2.70 and 2.71. In DOS 2.70, the fourth field is a byte, while in DOS 2.71 it is a word, shifting the remaining field.

One or more fields in a data structure may be pointers to additional data structures. Such data structures are often described in other entries of the appendix; in this case, under Interrupt 2Bh Function 03h.

The final section of an entry is a list of related functions. For this example, the reader may wish to refer to Interrupt 2Bh Function 02h (which would be the "Set DWIM Interpreter Parameters" call) and Interrupt 2Fh Function AEh subfunction 01h (the COMMAND.COM installable command hook). In addition, Interrupt 21h Function 0Ah is related, but not included in this appendix because it is a documented call. All cross references to functions not listed in this appendix are italicized.

This can not be repeated often enough: because these calls are not officially documented, they may change from one version of DOS to the next, or quite simply be in error. Extra caution and testing are a must when using calls and data structures described in this appendix. Particular care is required where the information is known to be incomplete or uncertain (as indicated by italics).

Glossary of Abbreviations

ASCIZ A zero-terminated ASCII string, such as 'A','B','C',00h.

BPB The BIOS Parameter Block stores the low-level layout of a drive. *See also* INT 21/AH=53h.

CDS The Current Directory Structure for a drive stores the current directory, type, and other information about a logical drive. *See also* INT 21/AH=52h.

DPB The DOS Drive Parameter Block stores the description of the media layout for a logical drive, as well as some housekeeping information. *See also* INT 21/AH=1Fh and INT 21/AH=32h.

DPL The DOS Parameter List is used to pass arguments to SHARE and network functions. *See also* INT 21/AX=5D00h.

DTA The Disk Transfer Address indicates where functions which do not take an explicit data address will read or store data. Although the name implies that only disk accesses use this address, other functions use it as well.

FAT The File Allocation Table of a disk, which records the clusters that are in use.

FCB A file control block, which is used by DOS 1.x functions to record the state of an open file. *See also* INT 21/AH=13h.

IFS An Installable File System which allows non-DOS format media to be used by DOS. In most ways, an IFS is very similar to a networked drive, although an IFS would typically be local rather than remote.

JFT The Job File Table (also called Open File Table) stored in a program's PSP which translates handles into SFT numbers. *See also* INT 21/AH=26h.

NCB A Network Control Block used to pass requests to NETBIOS and receive status information from the NETBIOS handler.

PSP The Program Segment Prefix is a 256-byte data area prepended to a program when it is loaded. It contains the command line that the program was invoked with, and a variety of housekeeping information for DOS. *See also* INT 21/AH=26h.

SDA The DOS Swappable Data Area, containing all of the variables used internally by DOS to record the state of a function call in progress. *See also* INT 21/AX=5D06h and INT 21/AX=5D0Bh.

SFT A System File Table is a DOS-internal data structure used to maintain the state of an open file for the DOS 2+ handle functions, just as an FCB maintains the state for DOS 1.x functions. *See also* INT 21/AH=52h.

INT 15h Function 2000h DOS 3.x

PRINT.COM - DISABLE CRITICAL REGION FLAG
Specify that PRINT should not set the user flag when it enters a DOS function call.

Call with:
AX 2000h

See Also: INT 15/AX=2001h

INT 15h Function 2001h DOS 3.x

PRINT.COM - SET CRITICAL REGION FLAG

Specify a location which PRINT should use as a flag to indicate when it enters a DOS function call.

Call with:

AX	2001h
ES:BX	pointer to byte which is to be incremented while in a DOS call

See Also: INT 15/AX=2000h

INT 21h Function 13h DOS 1+

DELETE DISK FILE with FCB

Although documented, this function is included because the file control block contains undocumented fields, and because the function itself has an undocumented quirk.

Call with:

AH	13h
DS:DX	pointer to a File Control Block (see below) with its filename field filled with a template for the files to be deleted ('?' wildcards allowed).

Returns:

AL	00h file found
	FFh file not found

Notes:

- Under DOS 3+, the file is opened in compatibility mode for sharing purposes.
- This function deletes everything in the current directory (including subdirectories) and sets the first byte of each deleted file's name to 00h (entry never used) instead of E5h if it is called on an extended FCB with the filename set to all question marks and bits 0-4 of the attribute set (bits 1 and 2 for DOS 1.x). This may have originally been an optimization to minimize directory searching after a total deletion, but it can corrupt the filesystem under DOS 2+ because subdirectories are removed without deleting the files that they contain.

Format of File Control Block (FCB):

Offset	Size	Description
-7	BYTE	extended FCB if FFh
-6	5 BYTEs	reserved
-1	BYTE	file attribute if extended FCB
00h	BYTE	drive number (0=default, 1=A:, etc)

01h	8 BYTEs	blank-padded file name
09h	3 BYTEs	blank-padded file extension
0Ch	WORD	current block number
0Eh	WORD	logical record size
10h	DWORD	file size
14h	WORD	date of last write (see *INT 21/AX=5700h*)
16h	WORD	time of last write (see *INT 21/AX=5700h*)
18h	8 BYTEs	reserved (see below)
20h	BYTE	record within current block
21h	DWORD	random access record number (if record size is 64 bytes, high byte is omitted)

Format of reserved field for DOS 1.x:

Offset	Size	Description
18h	BYTE	bit 7: set if logical device
		bit 6: set if open
		bits 5-0: disk number or logical device ID
19h	WORD	absolute current cluster number
1Bh	WORD	starting cluster number
1Dh	WORD	relative current cluster number
1Fh	*BYTE*	*apparently unused*

Format of reserved field for DOS 2.x:

Offset	Size	Description
18h	BYTE	bit 7: set if logical device
		bit 6: set if open
		bits 5-0: unknown
19h	WORD	starting cluster number
1Bh	*WORD*	*unknown*
1Dh	*BYTE*	*unknown*
1Eh	*BYTE*	*unknown*
1Fh	*BYTE*	*unknown*

Format of reserved field for DOS 3.x:

Offset	Size	Description
18h	BYTE	number of system file table entry for file
19h	BYTE	attributes
		bits 7,6: 00 = SHARE.EXE not loaded, disk file
		01 = SHARE.EXE not loaded, character device

		10 = SHARE.EXE loaded, remote file
		11 = SHARE.EXE loaded, local file
	bits 5-0:	low six bits of device attribute word

SHARE.EXE loaded, local file

1Ah	WORD	starting cluster of file
1Ch	WORD	offset within SHARE of sharing record (see *INT 21/AH=52h*)
1Eh	BYTE	file attribute
1Fh	*BYTE*	*unknown*

SHARE.EXE loaded, remote file

1Ah	WORD	number of sector containing directory entry
1Ch	WORD	relative cluster within file of last cluster accessed
1Eh	BYTE	absolute cluster number of last cluster accessed
1Fh	*BYTE*	*unknown*

SHARE.EXE not loaded

1Ah	BYTE	(low byte of device attribute word AND 0Ch) OR open mode
1Bh	WORD	starting cluster of file
1Dh	WORD	number of sector containing directory entry
1Fh	BYTE	number of directory entry within sector

Note:
If the FCB is opened on a character device, the DWORD at 1Ah is set to the address of the device driver header, and then the BYTE at 1Ah is overwritten.

See Also: *INT 21/AH=3Dh*

INT 21h Function 18h DOS 1+

UNUSED

This function returns immediately, and appears to be for CP/M compatibility.

Call with:

AH	18h

Returns:

AL	00h

INT 21h Function 1Dh DOS 1+

UNUSED

This function returns immediately, and appears to be for CP/M compatibility.

Call with:

AH 1Dh

Returns:

AL 00h

INT 21h Function 1Eh DOS 1+

UNUSED

This function returns immediately, and appears to be for CP/M compatibility.

Call with:

AH 1Eh

Returns:

AL 00h

INT 21h Function 1Fh DOS 1+

GET DEFAULT DRIVE PARAMETER BLOCK

Return the address of a disk description table for the current drive.

Call with:

AH 1Fh

Returns:

AL 00h No Error
FFh Error

DS:BX pointer to drive parameter block (see below for DOS 1.x, INT 21/AH=32h for DOS 2+)

Note:
For DOS 2+, this function merely invokes function 32h with DL=00h.

Format of Eagle MSDOS 1.25 drive parameter block:

Offset	Size	Description
00h	BYTE	entry number
01h	BYTE	physical drive number
02h	WORD	bytes per sector
04h	BYTE	number of sectors per cluster - 1
05h	*BYTE*	*unknown*
06h	WORD	starting sector number of first FAT
08h	BYTE	number of copies of FAT
09h	WORD	number of directory entries
0Bh	WORD	number of first data sector
0Dh	WORD	number of clusters on disk
0Fh	BYTE	sectors per FAT
10h	WORD	starting sector of directory
12h	WORD	address of allocation table

See Also: INT 21/AH=32h

INT 21h Function 20h DOS 1+

UNUSED

This function returns immediately, and appears to be for CP/M compatibility.

Call with:
AH 20h

Returns:
AL 00h

INT 21h Function 26h DOS 1+

CREATE PROGRAM SEGMENT PREFIX

Although documented, this function is included because the PSP which is created contains undocumented fields.

Call with:

AH	26h
DX	segment number at which to set up PSP

Returns:

The current PSP is copied to the specified segment.

Notes:

- The new PSP is updated with memory size information; INTs 22h, 23h, 24h are taken from the interrupt vector table.
- Under DOS 2+, DOS assumes that the caller's CS is the same as the segment of the PSP to copy.

Format of PSP:

Offset	Size	Description
00h	2 BYTEs	program exit point (INT 20h instruction)
02h	WORD	memory size in paragraphs
04h	BYTE	unused
05h	5 BYTEs	CP/M entry point (FAR jump to 000C0h) BUG: the jump address is 2 bytes too low for DOS 2+ on PSPs created by the EXEC function
06h	WORD	CP/M compatibility—size of first segment for .COM files
0Ah	DWORD	terminate address (old INT 22h)
0Eh	DWORD	break address (old INT 23h)
12h	DWORD	critical error handler (old INT 24h)
16h	WORD	parent PSP segment
18h	20 BYTEs	DOS 2+ open file table, FFh=unused
2Ch	WORD	DOS 2+ environment segment (see below)
2Eh	DWORD	DOS 2+ process's SS:SP on entry to last INT 21h call
32h	WORD	DOS 3+ max open files
34h	DWORD	DOS 3+ open file table address
38h	DWORD	DOS 3+ pointer to previous PSP (default FFFFFFFFh in 3.x)

		used by SHARE in DOS 3.3
3Ch	20 BYTEs	unused by DOS versions <= 4.01
50h	3 BYTEs	DOS function dispatcher (FAR routine)—CDh 21h CBh
53h	9 BYTEs	unused
5Ch	16 BYTEs	FCB #1 (see INT 21/AH=13h), filled in from first commandline argument (when opened, overwrites following FCB)
6Ch	20 BYTEs	FCB #2 (see INT 21/AH=13h), filled in from second commandline argument (when opened, overwrites part of command tail)
80h	128 BYTEs	command tail / default DTA buffer the command tail consists of a BYTE for the length, N BYTEs for the tail, followed by a BYTE containing 0Dh

Notes:

- For DOS versions 3.0 and up, the limit on simultaneously open files may be increased by allocating memory for a new open file table, filling it with FFh, copying the first 20 bytes from the default table, and adjusting the pointer and count at 34h and 32h. However, DOS versions through at least 3.30 will only copy the first 20 file handles into a child PSP (including the one created on EXEC).
- Many network redirectors, including those based on the original MS-Net implementation, use values of 80h-FEh in the open file table to indicate remote files.

Format of environment block:

Offset	Size	Description
00h	N BYTEs	first environment variable, ASCIZ string of form "var=value"
	N BYTEs	second environment variable, ASCIZ string
		...
	N BYTEs	last environment variable, ASCIZ string of form "var=value"
	BYTE	00h
DOS 3+		
	WORD	number of strings following environment (normally 1)
	N BYTEs	ASCIZ full pathname of program owning this environment other strings may follow

See Also: INT 21/AH=50h, INT 21/AH=51h, INT 21/AH=55h, *INT 21/AH=62h*, INT 21/AH=67h

INT 21h Function 32h DOS 2+

GET DRIVE PARAMETER BLOCK

Return the address of a disk description table for the specified drive.

Call with:

AH	32h
DL	drive number 0=default, 1=A:, etc.

Returns:

AL	FFh if invalid drive number else
DS:BX	pointer to drive parameter block (see below)

Note:

The OS/2 compatibility box supports the DOS 3.3 version of this call with the exception of the DWORD at offset 12h.

Format of DOS Drive Parameter Block:

Offset	Size	Description
00h	BYTE	drive number (0=A:, etc.)
01h	BYTE	unit number within device driver
02h	WORD	number of bytes per sector
04h	BYTE	largest sector number in cluster (one less than sectors/cluster)
05h	BYTE	shift count (log base 2) of the cluster size
06h	WORD	number of reserved (boot) sectors
08h	BYTE	number of copies of the FAT
09h	WORD	maximum number of root directory entries
0Bh	WORD	first data sector on medium
0Dh	WORD	largest possible cluster number (one more than the number of data clusters)
DOS 2.x		
0Fh	BYTE	number of sectors in one FAT copy
10h	WORD	first sector of root directory
12h	DWORD	address of device driver for this drive
16h	BYTE	media descriptor byte for medium
17h	BYTE	FFh indicates block must be rebuilt

18h	DWORD	address of next device block, offset=FFFFh indicates last
1Ch	WORD	starting cluster of current directory (0=root directory)
1Eh	64 BYTEs	ASCIZ current directory path string

DOS 3.x

0Fh	BYTE	number of sectors in one FAT copy
10h	WORD	first sector of root directory
12h	DWORD	address of device driver for this drive
16h	BYTE	media descriptor byte for medium
17h	BYTE	FFh if block must be rebuilt, 00h if block accessed
18h	DWORD	address of next device block, offset=FFFFh indicates last
1Ch	WORD	cluster at which to start search for free space when writing
1Eh	WORD	number of free clusters on drive, FFFFh if not known

DOS 4.0

0Fh	WORD	number of sectors in one FAT copy
11h	WORD	first sector of root directory
13h	DWORD	address of device driver for this drive
17h	BYTE	media descriptor byte for medium
18h	BYTE	FFh if block must be rebuilt, 00h if block accessed
19h	DWORD	address of next device block, offset=FFFFh indicates last
1Dh	WORD	cluster at which to start search for free space when writing
1Fh	WORD	number of free clusters on drive, FFFFh if not known

See Also: INT 21/AH=1Fh

INT 21h Function 3302h DOS 3+

GET AND SET EXTENDED CONTROL-BREAK CHECKING STATE

Set a new state for the extended Control-Break checking flag and return its old value.

Call with:

AX	3302h
DL	new state (00h for OFF or 01h for ON)

Returns:

DL	old state of extended BREAK checking

Note:
This function does not use any of the DOS-internal stacks and is thus fully reentrant.

INT 21h Function 34h DOS 2+

RETURN Critical Section Flag (InDOS) POINTER

Get the address of a flag which indicates when code within DOS is being executed, and it is thus unsafe to call DOS functions.

Call with:

AH 34h

Returns:

ES:BX pointer to 1-byte DOS "Critical Section Flag", also known as InDOSflag

Notes:

- When the critical section flag is nonzero, code within DOS is being executed. It is safe to enter DOS when both the critical section flag and the critical error flag are zero.
- The critical error flag is the byte after the critical section flag in DOS 2.x, and the byte BEFORE the critical section flag in DOS 3.x (except COMPAQ DOS 3.0, where the critical error flag is located 1AAh bytes BEFORE the critical section flag).
- For DOS 3+, an undocumented call exists to get the address of the critical error flag (see INT 21/AX=5D06h).

INT 21h Function 3700h DOS 2+

GET SWITCHAR

Return the character which is used to introduce command switches.

Call with:

AX 3700h

Returns:

AL FFh unsupported subfunction
else
DL current switch character

Notes:

- This call is documented in some OEM versions of some releases of DOS.

- This function is supported by the OS/2 v1.1 compatibility box.
- Although supported by DOS 4, COMMAND.COM and the DOS external commands ignore the value of the switch character.

See Also: INT 21/AX=3701h

INT 21h Function 3701h DOS 2+

SET SWITCHAR

Set a new value for the character which is used to introduce command switches.

Call with:

AX	3701h
DL	new switch character

Returns:

AL	FFh if unsupported subfunction

Notes:

- This call is documented in some OEM versions of some releases of DOS.
- This function is supported by the OS/2 v1.1 compatibility box.

See Also: INT 21/AX=3700h

INT 21h Functions 3702h, 3703h DOS 2.x and 4.0 only

AVAILDEV

Get or set the state of the flag which makes a \DEV\ prefix to character device names mandatory.

Call with:

AH	37h
AL	subfunction
	02h read device availability (as set by AL=03h)
	Returns:

DL device availability (always FFh under DOS 4.0)

03h set device availability

Call with:

DL 00h means \DEV\ must preceed device names

DL 00h means \DEV\ need not preceed device names

Returns:

AL FFh if invalid subfunction

Notes:

- All versions of DOS from 2.00 allow \DEV\ to be prepended to device names without generating an error even if the directory \DEV does not actually exist (other paths generate an error if they do not exist).
- Although DOS 4.0 accepts these calls, they have no effect.

INT 21h Function 41h DOS 3.1+

DELETE A FILE (UNLINK)

When invoked via INT 21/AX=5D00h, this function has the undocumented behavior of allowing wildcards in the filename, deleting all matching files.

Call with:

AH 41h

DS:DX pointer to ASCIZ pathname of file to delete

Returns:

CF set on error

AX error code (02h,05h) (see *INT 21/AH=59h*)

Note:

When invoked via INT 21/AX=5D00h, the filespec must be canonical (as returned by INT 21/AH=60h).

See Also: INT 21/AH=13h, INT 21/AX=5D00h, INT 21/AH=60h, INT 2F/AX=1113h

INT 21h Function 4400h — DOS 2+

GET DEVICE INFORMATION

Although documented, this function has a number of undocumented attribute bits.

Call with:

AX 4400h
BX file or device handle

Returns:

CF set on error
    AX error code (see *INT 21/AH=59h*)
CF clear if succesful
    DX device info
        character device if bit 7 set
            bit 0: console input device (standard input)
            bit 1: console output device (standard output)
            bit 2: current NUL device
            bit 3: CLOCK$ device
            bit 4: device uses INT 29h for single-char output
            bit 5: binary (raw) mode
            bit 6: not at EOF
            bit 7: set
            bit 11: media not removable
            bit 12: network device (DOS 3+)
            bit 14: can process IOCTL control strings (see *INT21/AX=4402h*)
        file if bit 7 clear
            bits 0-5: block device number
            bit 6: file has not been written to
            bit 7: clear
            bit 11: media not removable
            bit 12: network device (DOS 3+)
            bit 14: don't set file date/time on closing (DOS 4+)
            bit 15: file is remote (DOS 3+)

See Also: INT 2F/AX=122Bh

INT 21h Function 4Ah DOS 2+

ADJUST MEMORY BLOCK SIZE (SETBLOCK)

Although documented, this function is included because of its undocumented behavior when there is not enough memory to satisfy the request.

Call with:

AH	4Ah
ES	segment address of block to change
BX	new size in paragraphs

Returns:

CF set on error

AX	error code (07h,08h,09h) (see AH=59h)
BX	maximum size possible for the block (if AX=08h)

Note:

Under PCDOS 2.1 and DOS 3.x, if there is insufficient memory to expand the block as much as requested, the block will still be made as large as possible (value in BX).

See Also: INT 21/AH=48h, INT 21/49h

INT 21h Functions 4B01h, 4B04h DOS 2+

LOAD OR EXECUTE (EXEC)

In addition to the documented subfunctions to execute a child process and load an overlay, the EXEC function also has an undocumented subfunction to load and relocate a program without beginning execution. There are also reports that some versions of DOS may support a second undocumented subfunction to run a child process in the background.

Call with:

AH	4Bh
AL	subfunction
	01h load but do not execute
	04h Reportedly called by MSC spawn(P_NOWAIT,...) when running under DOS 4.x.

Returns unsuccessfully under DOS 4.0, but may be successful in the original European OEM MSDOS 4.0, which has limited multitasking built in.

DS:DX pointer to ASCIZ filename
ES:BX pointer to parameter block (see below)

Returns:

CF set on error
AX error code (01h,02h,05h,08h,0Ah,0Bh) (see *INT 21/AH=59h*)
CF clear if successful
if subfunction 01h, process ID set to new program's PSP; get with INT 21/AH=51h or *INT 21/AH=62h*

Notes:

- DOS 2.x destroys all registers, including SS:SP.
- DOS 3.0+ destroys BX and DX.
- The calling process must ensure that there is enough unallocated memory available; if necessary, by releasing memory with *INT 21/AH=49h* or INT 21/AH=4Ah.

Format of EXEC parameter block for AL=01h:

Offset	Size	Description
00h	WORD	segment of environment (0 to use current) (see INT 21/AH=26h)
02h	DWORD	pointer to command line
06h	DWORD	pointer to first FCB (see INT 21/AH=13h)
0Ah	DWORD	pointer to second FCB (see INT 21/AH=13h)
0Eh	DWORD	will hold subprogram's initial SS:SP on return
12h	DWORD	will hold entry point (CS:IP) on return

See Also: *INT 21/AH=4Ch, INT 21/AH=4Dh,* INT 2E

INT 21h Function 4Eh DOS 2+

FIND FIRST ASCIZ (FINDFIRST)

Although documented, this function is included because of the undocumented fields in its data structure, which are used to record the progress of the directory search, and because of an undocumented quirk, and a subtle bug.

Call with:

AH 4Eh
CX search attributes (see *INT 21/AX=4301h*)

bit 0 = read only
1 = hidden file
2 = system file
3 = volume label
4 = subdirectory
5 = written since backup ("archive" bit)
8 = shareable (Novell NetWare)

DS:DX pointer to ASCIZ filespec (drive, path, and wildcards allowed)

Returns:

CF set on error
AX error code (02h,12h) (see *INT 21/AH=59h*)

CF clear if successful
[DTA] data block (see below)

Notes:

- For search attributes other than 08h, all files with at MOST the specified attribute bits, the archive (20h) bit, and the read-only (01h) bit set will be returned. Under DOS 2.x, searching for attribute 08h (volume label) will also return normal files, while DOS 3+ returns only the volume label (if any).
- If the given filespec is the name of a character device without wildcards, the call will return successfully. DOS 2.x returns attribute 00h, size 0, and the current date and time. DOS 3+ returns attribute 40h and the current date and time.

BUG:

Under DOS 3.x and 4.x, the second and subsequent calls to this function with a character device name (no wildcards) and search attributes which include the volume-label bit (08h) will fail unless there is an intervening DOS call which implicitly or explicitly performs a directory search without the volume-label bit. Such implicit searches are performed by CREATE (*INT 21/AH=3Ch*), OPEN (*INT 21/AH=3Dh*), UNLINK (INT 21/AH=41h), and RENAME (INT 21/AH=56h).

Format of FindFirst data block:

Offset	Size	Description
PCDOS 3.10/4.01, MSDOS 3.2/3.3		
00h	BYTE	drive letter
01h	11 BYTEs	search template
0Ch	BYTE	search attributes

DOS 2.x and some DOS 3.x

00h	BYTE	search attributes
01h	BYTE	drive letter
02h	11 BYTEs	search template

DOS 2.x and most 3.x

0Dh	WORD	entry count within directory
0Fh	*DWORD*	*pointer to DTA*
13h	WORD	cluster number of start of parent directory

PCDOS 4.01, MSDOS 3.2/3.3

0Dh	WORD	entry count within directory
0Fh	WORD	cluster number of start of parent directory
11h	4 BYTEs	reserved

all versions, documented fields

15h	BYTE	attribute of file found
16h	WORD	file time
		bits 11-15: hour
		bits 5-10: minute
		bits 0-4: seconds/2
18h	WORD	file date
		bits 9-15: year-1980
		bits 5-8: month
		bits 0-4: day
1Ah	DWORD	file size
1Eh	13 BYTEs	ASCIZ filename+extension

See Also: *INT 21/AH=11h, INT 21/AH=4Fh,* INT 2F/AX=111Bh

INT 21h Function 50h DOS 2+

SET PSP SEGMENT

Force a new value for DOS's record of the current process's PSP segment, thus effectively becoming another process.

Call with:

AH	50h
BX	segment address of new PSP (see INT 21/AH=26h for format)

Notes:

- Under DOS 2.x, this function cannot be invoked inside an INT 28h handler without setting the Critical Error flag, because it uses the same stack which DOS is using at the time of the INT 28h.
- Under DOS 3+, this function does not use any of the DOS-internal stacks and is thus fully reentrant.
- This function is supported by the OS/2 v1.1 compatibility box.

See Also: INT 21/AH=26h, INT 21/AH=51h, *INT 21/AH=62h*

INT 21h Function 51h DOS 2+

GET PSP SEGMENT

Return the segment address of the current process's PSP, which is used by DOS as a process identifier.

Call with:

AH 51h

Returns:

BX current PSP segment (see INT 21/AH=26h for format)

Notes:

Under DOS 2.x, this function cannot be invoked inside an INT 28h handler without setting the Critical Error flag, because it uses the same stack which DOS is using at the time of the INT 28h.
Under DOS 3+, this function does not use any of the DOS-internal stacks and is thus fully reentrant.
The documented INT 21/AH=62h is identical to this call, but only available with DOS 3+.
In DOS 3+, the two functions are identical, jumping to the same code (which simply loads BX from the SDA current PSP field).
This function is supported by the OS/2 v1.1 compatibility box.

See Also: INT 21/AH=26h, INT 21/AH=50h, *INT 21/AH=62h*

INT 21h Function 52h DOS 2+

GET LIST OF LISTS

Return the address of DOS's internal list of tables and lists. Most internal data structures are reachable through this list.

Call with:

AH 52h

Returns:

ES:BX pointer to DOS list of lists

Note:

This call is partially supported by the OS/2 v1.1 compatibility box. Most pointers in the returned list are FFFFh:FFFFh, however.

Format of List of Lists:

Offset	Size	Description
-12	WORD	(DOS 3.1-3.3) sharing retry count (see *INT 21/AX=440Bh*)
-10	WORD	(DOS 3.1-3.3) sharing retry delay (see *INT 21/AX=440Bh*)
-8	DWORD	(DOS 3.x) pointer to current disk buffer
-4	WORD	(DOS 3.x) pointer in DOS code segment of unread CON input When CON is read via a handle, DOS reads an entire line, and returns the requested portion, buffering the rest for the next read. 0000h indicates no unread input
-2	WORD	segment of first memory control block
00h	DWORD	pointer to first DOS Drive Parameter Block (see INT 21/AH=32h)
04h	DWORD	pointer to list of DOS file tables (see below)
08h	DWORD	pointer to CLOCK$ device driver, resident or installable
0Ch	DWORD	pointer to CON device driver, resident or installable
DOS 2.x		
10h	BYTE	number of logical drives in system
11h	WORD	maximum bytes/block of any block device
13h	DWORD	pointer to first disk buffer (see below)
17h	18 BYTEs	actual NUL device driver header (not a pointer!) This is the first device on DOS's linked list of device drivers. (see below for format)

DOS 3.0

10h	BYTE	number of block devices
11h	WORD	maximum bytes/block of any block device
13h	DWORD	pointer to first disk buffer (see below)
17h	DWORD	pointer to array of current directory structures (see below)
1Bh	BYTE	value of LASTDRIVE command in CONFIG.SYS (default 5)
1Ch	DWORD	pointer to STRING= workspace area
20h	WORD	size of STRING area (the x in STRING=x from CONFIG.SYS)
22h	DWORD	pointer to FCB table
26h	WORD	the y in FCBS=x,y from CONFIG.SYS
28h	18 BYTEs	actual NUL device driver header (not a pointer!) This is the first device on DOS's linked list of device drivers. (see below for format)

DOS 3.1-3.3

10h	WORD	maximum bytes/block of any block device
12h	DWORD	pointer to first disk buffer (see below)
16h	DWORD	pointer to array of current directory structures (see below)
1Ah	DWORD	pointer to FCB table (if CONFIG.SYS contains FCBS=)
1Eh	WORD	number of protected FCBs (the y in FCBS=x,y)
20h	BYTE	number of block devices
21h	BYTE	value of LASTDRIVE command in CONFIG.SYS (default 5)
22h	18 BYTEs	actual NUL device driver header (not a pointer!) This is the first device on DOS's linked list of device drivers. (see below for format) (see also INT 2F/AX=122Ch)
34h	BYTE	number of JOIN'ed drives

DOS 4.x

10h	WORD	maximum bytes/block of any block device
12h	DWORD	pointer to disk buffer info (see below)
16h	DWORD	pointer to array of current directory structures (see below)
1Ah	DWORD	pointer to FCB table (if CONFIG.SYS contains FCBS=)
1Eh	WORD	number of protected FCBs (the y in FCBS=x,y)
20h	BYTE	number of block devices
21h	BYTE	value of LASTDRIVE command in CONFIG.SYS (default 5)
22h	18 BYTEs	actual NUL device driver header (not a pointer!) This is the first device on DOS's linked list of device drivers. (see below for format) (see also INT 2F/AX=122Ch)
34h	*BYTE*	*number of JOIN'ed drives*

35h	WORD	pointer within IBMDOS code segment to list of special program names (see below)
37h	DWORD	pointer to FAR routine for resident IFS utility functions (see below) This routine may be called by any IFS driver which does not wish to service IFS functions 20h or 24h-28h itself.
3Bh	DWORD	pointer to chain of IFS (installable file system) drivers
3Fh	WORD	the x in BUFFERS x,y (rounded up to multiple of 30 if in EMS)
41h	WORD	the y in BUFFERS x,y
43h	BYTE	boot drive (1=A:)
44h	*BYTE*	*unknown*
45h	WORD	extended memory size in K

Format of memory control block:

Offset	Size	Description
00h	BYTE	block type: 5Ah if last block in chain, otherwise 4Dh
01h	WORD	PSP segment of owner, 0000h if free, 0008h if belongs to DOS
03h	WORD	size of memory block in paragraphs
05h	3 BYTEs	unused
DOS 2.x, 3.x		
08h	8 BYTEs	unused
DOS 4.x		
08h	8 BYTEs	ASCII program name if PSP memory block, else garbage null-terminated if less than 8 characters

Notes:

- Under DOS 3.1+, the first memory block is the DOS data segment, containing installable drivers, buffers, etc.
- Under DOS 4.x, the first memory block is divided into subsegments, each with its own memory control block (see below), the first of which is at offset 0000h.

Format of DOS 4.x data segment subsegment control blocks:

Offset	Size	Description
00h	BYTE	subsegment type (blocks typically appear in this order)
		"D" device driver
		"E" device driver appendage
		"I" IFS (Installable File System) driver
		"F" FILES= control block storage area (for FILES > 5)
		"X" FCBS= control block storage area, if present
		"C" BUFFERS EMS workspace area (if BUFFERS /X option used)

		"B" BUFFERS= storage area
		"L" LASTDRIVE= current directory structure array storage area
		"S" STACKS= code and data area, if present (see below)
01h	WORD	paragraph of subsegment start (usually the next paragraph)
03h	WORD	size of subsegment in paragraphs
05h	3 BYTEs	unused
08h	8 BYTEs	for types "D" and "I", base name of file from which the driver was loaded (unused for other types)

Format of data at start of STACKS code segment (if present):

Offset	Size	Description
00h	*WORD*	*unknown*
02h	WORD	number of stacks (the x in STACKS=x,y)
04h	WORD	size of stack control block array (should be 8*x)
06h	WORD	size of each stack (the y in STACKS=x,y)
08h	DWORD	pointer to STACKS data segment
0Ch	WORD	offset in STACKS data segment of stack control block array
0Eh	WORD	offset in STACKS data segment of last element of that array
10h	WORD	offset in STACKS data segment of the entry in that array for the next stack to be allocated (initially same as value in 0Eh and works its way down in steps of 8 to the value in 0Ch as hardware interrupts pre-empt each other)

Note:

The STACKS code segment data may, if present, be located as follows:

- DOS 3.2: The code segment data is at a paragraph boundary fairly early in the IBMBIO segment (seen at 0070:0190h).
- DOS 3.3: The code segment is at a paragraph boundary in the DOS data segment, which may be determined by inspecting the segment pointers of the vectors for those of interrupts 02h, 08h-0Eh, 70h, 72-77h which have not been redirected by device drivers or TSRs.
- DOS 4.x: Identified by sub-segment control block type "S" within the DOS data segment.

Format of array elements in STACKS data segment:

Offset	Size	Description
00h	BYTE	status: 00h=free, 01h=in use, 03h=corrupted by overflow of higher stack.
01h	BYTE	not used
02h	WORD	previous SP
04h	WORD	previous SS

06h WORD pointer to word at top of stack (new value for SP).

Note:
The word at the top of the stack is preset to point back to this control block.

SHARE.EXE hooks (DOS 3.1-4.01):

(offsets from first system file table—pointed at by ListOfLists+04h)

Offset	Size	Description
-3Ch	*DWORD*	*pointer to unknown FAR routine* **Note:** not called by MSDOS 3.3, set to 0000h:0000h by SHARE 3.3
-38h	DWORD	pointer to FAR routine called on opening file **Call with:** SS set to DOS CS DS set to DOS CS SDA first filename pointer points at filename (see INT 21/AX=5D06h) **Returns:** CF clear if successful CF set on error AX DOS error code (24h) (see *INT 21/AH=59h*)
-34h	DWORD	pointer to FAR routine called on closing file **Call with:** ES:DI pointer to system file table SS set to DOS CS additional arguments (if any) unknown **Note:** does something to every lock record for the file
-30h	DWORD	pointer to FAR routine to close all files for given computer (called by AX=5D03h) **Note:** SHARE assumes SS=DOS CS, directly accesses DOS internals
-2Ch	DWORD	pointer to FAR routine to close all files for given process (called by AX=5D04h) **Note:** SHARE assumes SS=DOS CS, directly accesses DOS internals
-28h	DWORD	pointer to FAR routine to close file by name (called by AX=5D02h) **Call with:** DS:SI pointer to DOS parameter list (see INT 21/AX=5D00h) DPL's DS:DX pointer to name of file to close

SS set to DOS CS

Returns:

CF clear if successful

CF set on error

AX DOS error code (03h) (see *INT 21/AH=59h*)

Note:

SHARE directly accesses DOS internals

-24h DWORD pointer to FAR routine to lock region of file

Call with:

BX file handle

CX:DX starting offset

SI:AX size

SS set to DOS CS

Returns:

CF set on error

AL DOS error code (21h) (see *INT 21/AH=59h*)

Notes:

only called if file is marked as remote

SHARE directly accesses DOS internals

-20h DWORD pointer to FAR routine to unlock region of file

Call with:

BX file handle

CX:DX starting offset

SI:AX size

SS set to DOS CS

Returns:

CF set on error

AL DOS error code (21h) (see *INT 21/AH=59h*)

Notes:

only called if file is marked as remote

SHARE directly accesses DOS internals

-1Ch DWORD pointer to FAR routine to check if file region is locked

Call with:

ES:DI pointer to system file table entry for file

CX length of region from current position in file

SS set to DOS CS

Returns:

CF set if any portion of region locked

AX 0021h

Note:
SHARE directly accesses DOS internals

-18h DWORD pointer to FAR routine to get open file list entry (called by AX=5D05h)

Call with:
DS:SI pointer to DOS parameter list (see INT 21/AX=5D00h)
DPL's BX index of sharing record
DPL's CX index of SFT in SFT chain of sharing rec
SS set to DOS CS

Returns:
CF set on error or not loaded
AX DOS error code (12h) (see *INT 21/AH=59h*)
CF clear if successful
ES:DI pointer to filename
CX number of locks owned by specified SFT
BX network machine number
DX destroyed

Note:
SHARE directly accesses DOS internals

-14h DWORD pointer to FAR routine for updating FCB from SFT

Call with:
DS:SI pointer to unopened FCB
ES:DI pointer to system file table entry

Returns:
BL C0h

Note:
copies the following fields from SFT to FCB: starting cluster of file, sharing record offset, and file attribute

-10h DWORD pointer to FAR routine to get first cluster of FCB file

Call with:
ES:DI pointer to system file table entry
DS:SI pointer to FCB

Returns:
CF set if SFT closed or sharing record offsets mismatched
CF clear if successful
BX starting cluster number from FCB

-0Ch DWORD pointer to FAR routine to close file if duplicate for process

Call with:
DS:SI pointer to system file table
SS set to DOS CS

Returns:

AX number of handle in JFT which already uses SFT

Notes:

called during open/create of a file
SHARE directly accesses DOS internals
if the SFT was opened with inheritance enabled and sharing mode 111, this call does something to all other SFTs owned by the same process which have the same file open mode and sharing record

-08h *DWORD* *pointer to unknown FAR routine*

Call with:

SS set to DOS CS
DS set to DOS CS

Notes:

SHARE directly accesses DOS internals
closes various handles referring to the file most-recently opened

-04h DWORD pointer to FAR routine to update directory info in related SFT entries

Call with:

ES:DI pointer to system file table entry for file (see below)
AX subfunction (apply to each related SFT)
00h: update time stamp (offset 0Dh) and date stamp (offset 0Fh)
01h: update file size (offset 11h) and starting cluster (offset 0Bh). Sets last-accessed cluster fields to start of file if file never accessed
02h: as function 01h, but last-accessed fields always changed
03h: do both functions 00h and 02h

Note:

follows ptr at offset 2Bh in system file table entries
NOP if opened with no-inherit or via FCB

Format of sharing record:

Offset	Size	Description
00h	BYTE	flag 00h free block 01h allocated block FFh end marker
01h	WORD	size of block
03h	BYTE	checksum of pathname (including NUL) if sum of ASCII values is N, checksum is (N/256 + N%256)

04h	WORD	offset in SHARE's DS of lock record (see below)
06h	DWORD	pointer to start of system file table chain for file
0Ah	WORD	unique sequence number
0Ch	var	ASCIZ full pathname

Format of SHARE.EXE lock record:

Offset	Size	Description
00h	WORD	offset in SHARE's DS of next lock table in list
02h	DWORD	offset in file of start of locked region
06h	DWORD	offset in file of end of locked region
0Ah	DWORD	pointer to System File Table entry for this file
0Eh	WORD	PSP segment of lock's owner

Format of DOS 2.x system file tables:

Offset	Size	Description
00h	DWORD	pointer to next file table
04h	WORD	number of files in this table
06h	28h bytes per file	

Offset	Size	Description
00h	BYTE	number of file handles referring to this file
01h	BYTE	file open mode (see *INT 21/AH=3Dh*)
02h	BYTE	file attribute
03h	BYTE	drive (0=character device, 1=A:, 2=B:, etc)
04h	11 BYTEs	filename in FCB format (no path,no period,blank-padded)
0Fh	*WORD*	*unknown*
11h	*WORD*	*unknown*
13h	*DWORD*	*file size*
17h	WORD	file date in packed format (see *INT 21/AX=5700h*)
19h	WORD	file time in packed format (see *INT 21/AX=5700h*)
1Bh	BYTE	device attribute (see INT 21 / AX=4400h)

character device

1Ch	DWORD	pointer to device driver

block device

1Ch	WORD	starting cluster of file
1Eh	WORD	relative cluster in file of last cluster accessed
20h	WORD	absolute cluster number of current cluster
22h	*WORD*	*unknown*
24h	*DWORD*	*current file position*

Format of DOS 3.1 and higher system file tables and FCB tables:

Offset	Size	Description
00h	DWORD	pointer to next file table
04h	WORD	number of files in this table
06h	35h bytes per file	

Offset	Size	Description
00h	WORD	number of file handles referring to this file
02h	WORD	file open mode (see *INT 21/AH=3Dh*) bit 15 set if this file opened via FCB
04h	BYTE	file attribute
05h	WORD	device info word (see INT 21/AX=4400h); includes drive number
07h	DWORD	pointer to device driver header if character device else pointer to DOS Drive Parameter Block (see INT 21/AH=32h)
0Bh	WORD	starting cluster of file
0Dh	WORD	file time in packed format (see *INT 21/AX=5700h*)
0Fh	WORD	file date in packed format (see *INT 21/AX=5700h*)
11h	DWORD	file size
15h	DWORD	current offset in file
19h	WORD	relative cluster within file of last cluster accessed
1Bh	WORD	absolute cluster number of last cluster accessed *0000h if file never read or written*
1Dh	WORD	number of sector containing directory entry
1Fh	BYTE	number of directory entry within sector (byte offset/32); WORD (not BYTE) in DOS 3.0 only. All subsequent fields shifted down in DOS 3.0 only.
20h	11 BYTEs	filename in FCB format (no path/period, blank-padded)
2Bh	DWORD	(SHARE.EXE) pointer to previous SFT sharing same file
2Fh	WORD	(SHARE.EXE) network machine number which opened file
31h	WORD	PSP segment of file's owner (see INT 21/AH=26h), except for AUX, CON, and PRN, which hold effective PSP left over from INIT
33h	WORD	offset within SHARE.EXE code segment of sharing record (see below) 0000h if none

Format of DOS 4.x system file tables and FCB tables:

Offset	Size	Description
00h	DWORD	pointer to next file table
04h	WORD	number of files in this table
06h	3Bh bytes per file	

Offset	Size	Description
00h	WORD	number of file handles referring to this file
02h	WORD	file open mode (see *INT 21/AH=3Dh*)

		bit 15 set if this file opened via FCB
04h	BYTE	file attribute
05h	WORD	device info word (see INT 21/AX=4400h)
		bit 15 set if remote file
		bit 14 set means do not set file date/time on closing
07h	DWORD	pointer to device driver header if character device
		else pointer to DOS Drive Parameter Block (see INT 21/AH=32h)
		or REDIR data
0Bh	WORD	starting cluster of file
0Dh	WORD	file time in packed format (see *INT 21/AX=5700h*)
0Fh	WORD	file date in packed format (see *INT 21/AX=5700h*)
11h	DWORD	file size
15h	DWORD	current offset in file

local file

19h	WORD	relative cluster within file of last cluster accessed
1Bh	DWORD	number of sector containing directory entry
1Fh	BYTE	number of directory entry within sector (byte offset/32)

network redirector

19h	DWORD	pointer to REDIRIFS record
1Dh	*3 BYTEs*	*unknown*

20h	11 BYTEs	filename in FCB format (no path/period, blank-padded)
2Bh	DWORD	(SHARE.EXE) pointer to previous SFT sharing same file
2Fh	WORD	(SHARE.EXE) network machine number which opened file
31h	WORD	PSP segment of file's owner (see INT 21/AH=26h)
33h	WORD	offset within SHARE.EXE code segment of sharing record
		(see below) 0000h if none
35h	WORD	(local) absolute cluster number of last cluster accessed
		(redirector) unknown
37h	DWORD	pointer to IFS driver for file, 0000000h if native DOS

Format of current directory structure (array, 51h bytes [58h for DOS 4.x] per drive):

Offset	Size	Description
00h	67 BYTEs	current path as ASCIZ, usually starting with 'x:\' or '\\machine\'
43h	WORD	bit flags
		bit 15: network drive \ installable file system if both set
		bit 14: physical drive / invalid drive if neither bit set

		bit 13: JOIN'ed, current path is actual path without JOIN drive letter in path may differ from logical drive name bit 12: SUBST'ed, current path is actual path without SUBST drive letter in path may differ from logical drive name
45h	DWORD	pointer to Drive Parameter Block (DPB) for this drive
local drives		
49h	WORD	starting cluster of current directory 0000h for root directory, FFFFh if never accessed
4Bh	*WORD*	*unknown, seems always to be FFFFh*
4Dh	*WORD*	*unknown, seems always to be FFFFh*
network drives		
49h	DWORD	pointer to a redirector/REDIRIFS record, else FFFFFFFFh
4Dh	*WORD*	*stored parameter from INT 21/AX=5F03h*
all		
4Fh	WORD	Offset of '\' in current path field representing root directory of logical drive (2 if not SUBST'ed or JOIN'ed, otherwise number of bytes in SUBST/JOIN path)
DOS 4.x		
51h	*BYTE*	*unknown, used by network*
52h	DWORD	pointer to IFS driver for this drive, 00000000h if native DOS
56h	*WORD*	*unknown*

Format of device driver header:

Offset	Size	Description
00h	DWORD	pointer to next driver, offset=FFFFh if last driver
04h	WORD	device attributes

Character device:

bit 15 set
bit 14 IOCTL supported (see *INT 21/AH=44h*)
bit 13 (DOS 3+) output until busy supported
bit 12 reserved
bit 11 (DOS 3+) OPEN/CLOSE/RemMedia calls supported
bits 10-7 reserved
bit 6 (DOS 3.2+) Generic IOCTL call supported (command 13h)
(see *INT 21/AX=440Ch, INT 21/AX=440Dh*)
bit 5 reserved
bit 4 device is special (use INT 29 "fast console output")
bit 3 device is CLOCK$ (all reads/writes use transfer record described below)

bit 2 device is NUL
bit 1 device is standard output
bit 0 device is standard input

Block device:

bit 15 clear
bit 14 IOCTL supported
bit 13 non-IBM format
bit 12 reserved
bit 11 (DOS 3+) OPEN/CLOSE/RemMedia calls supported
bit 10 reserved
bit 9 unknown, set by DOS 3.3 DRIVER.SYS for "new" drives
bit 8 unknown, set by DOS 3.3 DRIVER.SYS for "new" drives
bit 7 reserved
bit 6 (DOS 3.2+) Generic IOCTL call supported (command 13h)
implies support for commands 17h and 18h (see *INT 21/AX=440Ch, INT 21/AX=440Dh, INT 21/AX=440Eh, INT 21/AX=440Fh)*
bits 5-2 reserved
bit 1 driver supports 32-bit sector addressing
bit 0 reserved

06h	WORD	device strategy entry point call with ES:BX pointer to request header (see INT 2F/AX=0802h)
08h	WORD	device interrupt entry point
character device		
0Ah	8 BYTEs	blank-padded character device name
block device		
0Ah	BYTE	number of subunits (drives) supported by this driver
0Bh	7 BYTEs	unused
12h	WORD	(CD-ROM driver) reserved, must be 0000h
14h	BYTE	(CD-ROM driver) drive letter (must initially be 00h)
15h	BYTE	(CD-ROM driver) number of units
16h	6 BYTEs	(CD-ROM driver) signature 'MSCDnn' where 'nn' is version (currently '00')

Format of CLOCK$ transfer record:

Offset	Size	Description
00h	WORD	number of days since 1-Jan-1980
02h	BYTE	minutes
03h	BYTE	hours

04h	BYTE	hundredths of second
05h	BYTE	seconds

Format of DOS 2.x disk buffer:

Offset	Size	Description
00h	DWORD	pointer to next disk buffer, offset=FFFFh if last least-recently used buffer is first in chain
04h	BYTE	drive (0=A:, 1=B:, etc), FFh if not in use
05h	*3 BYTEs*	*apparently unused (seems always to be 00h 00h 01h)*
08h	WORD	logical sector number
0Ah	BYTE	number of copies to write (1 for non-FAT sectors)
0Bh	BYTE	sector offset between copies if multiple copies to be written
0Ch	DWORD	pointer to DOS Drive Parameter Block (see INT 21/AH=32h)
10h		buffered data

Format of DOS 3.x disk buffer:

Offset	Size	Description
00h	DWORD	pointer to next disk buffer, offset=FFFFh if last least-recently used buffer is first in chain
04h	BYTE	drive (0=A:,1=B:, etc), FFh if not in use
05h	BYTE	flags *bit 7: unknown* bit 6: buffer dirty bit 5: buffer has been referenced *bit 4: unknown* bit 3: sector in data area bit 2: sector in a directory, either root or subdirectory bit 1: sector in FAT *bit 0: boot sector*
06h	WORD	logical sector number
08h	BYTE	number of copies to write (1 for non-FAT sectors)
09h	BYTE	sector offset between copies if multiple copies to be written
0Ah	DWORD	pointer to DOS Drive Parameter Block (see INT 21/AH=32h)
0Eh	*WORD*	*unused (almost always 0)*
10h		buffered data

Format of DOS 4.00 (before UR 25066 Corrective Services Disk) disk buffer info:

Offset	Size	Description
00h	DWORD	pointer to array of disk buffer hash chain heads (see below)
04h	WORD	number of disk buffer hash chains (referred to as NDBCH below)
06h	DWORD	pointer to lookahead buffer, zero if not present
0Ah	WORD	number of lookahead sectors, else zero (the y in BUFFERS=x,y)
0Ch	BYTE	00h if buffers in EMS (/x), FFh if not
0Dh	WORD	EMS handle for buffers, zero if not in EMS
0Fh	WORD	EMS physical page number used for buffers (usually 255)
11h	*WORD*	*apparently always 0001h*
13h	WORD	segment of EMS physical page frame
15h	*WORD*	*seems always to be zero*
17h	*4 WORDs*	*EMS partial page mapping information*

Format of DOS 4.01 (from UR 25066 Corrective Services Disk on) disk buffer info:

Offset	Size	Description
00h	DWORD	pointer to array of disk buffer hash chain heads (see below)
04h	WORD	number of disk buffer hash chains (referred to as NDBCH below)
06h	DWORD	pointer to lookahead buffer, zero if not present
0Ah	WORD	number of lookahead sectors, else zero (the y in BUFFERS=x,y)
0Ch	BYTE	01h, possibly to distinguish from pre-UR 25066 format (see above)
0Dh	*WORD*	*EMS segment for BUFFERS (only with /XD)*
0Fh	WORD	EMS physical page number of EMS segment above (only with /XD)
11h	*WORD*	*unknown EMS segment (only with /XD)*
13h	WORD	EMS physical page number of above (only with /XD)
15h	*BYTE*	*number of EMS page frames present (only with /XD)*
16h	WORD	segment of one-sector workspace buffer allocated in main memory if BUFFERS /XS or /XD options are in effect, possibly to avoid DMA into EMS
18h	WORD	EMS handle for buffers, zero if not in EMS
1Ah	WORD	EMS physical page number used for buffers (usually 255)
1Ch	*WORD*	*apparently always zero*
1Eh	WORD	segment of EMS physical page frame
20h	*WORD*	*apparetnly always zero*
22h	BYTE	00h if /XS, 01h if /XD, FFh if BUFFERS not in EMS.

Format of DOS 4.x disk buffer hash chain head (array, one entry per chain):

Offset	Size	Description
00h	WORD	EMS logical page number in which chain is resident, -1 if not in EMS
02h	DWORD	pointer to least recently used buffer header. All buffers on this chain are in the same segment.
06h	BYTE	number of dirty buffers on this chain
07h	BYTE	reserved (00h)

Notes:

- Buffered disk sectors are assigned to chain N where N is the sector's address modulo NDBCH (number disk buffer chain head), 0 <= N <= NDBCH-1.
- Each chain resides completely within one EMS page.
- This structure is in main memory even if the buffers are in EMS.

Format of DOS 4.x disk buffer:

Offset	Size	Description
00h	WORD	forward ptr, offset only, to next least recently used buffer
02h	WORD	backward ptr, offset only
04h	BYTE	drive (0=A,1=B, etc), FFh if not in use
05h	BYTE	flags bit 7: remote buffer bit 6: buffer dirty bit 5: buffer has been referenced bit 4: search data buffer (only valid if remote buffer) bit 3: sector in data area bit 2: sector in a directory, either root or subdirectory bit 1: sector in FAT bit 0: reserved
06h	DWORD	logical sector number
0Ah	BYTE	number of copies to write for FAT sectors, same as number of FATs for data and directory sectors, usually 1
0Bh	WORD	offset in sectors between copies to write for FAT sectors
0Dh	DWORD	pointer to DOS Drive Parameter Block (see INT 21/AH=32h)
11h	WORD	buffer use count if remote buffer (see flags above)
13h	BYTE	reserved
14h		buffered data

Notes:

- All buffered sectors which have the same hash value (computed as the sum of high and low words of the logical sector number divided by NDBCH) are on the same doubly-linked circular chain.
- The links between buffers consist of offset addresses only, the segment being the same for all buffers in the chain.

Format of IFS driver list:

Offset	Size	Description
00h	DWORD	pointer to next driver header
04h	*8 BYTEs*	*IFS driver name (blank padded), as used by FILESYS command*
0Ch	*4 Bytes unknown*	
10h	DWORD	pointer to IFS utility function entry point (see below)
		call with ES:BX pointing at IFS request (see below)
14h	WORD	offset in header's segment of driver entry point

additional fields (if any) unknown

Call IFS utility function entry point with:

AH 20h miscellaneous functions

AL 00h get date

Returns:

CX year
DH month
DL day

AL 01h get process ID and computer ID

Returns:

BX current PSP segment
DX active network machine number

AL 05h get file system info

Call with:

ES:DI pointer to 16-byte info buffer

Returns:

buffer filled

Offset	Size	Description
00h	2 BYTEs	unused
02h	WORD	number of SFT entries (actually counts only the first two file table arrays)
04h	WORD	number of FCB table entries
06h	WORD	number of proctected FCBs
08h	6 BYTEs	unused

0Eh WORD largest sector size supported

AL 06h get machine name

Call with:

ES:DI pointer to 18-byte buffer for name

Returns:

buffer filled with name starting at offset 02h

AL 08h get sharing retry count

Returns:

BX sharing retry count

AL other

Returns:

CF set

AH 21h get redirection state

BH = type (03h disk, 04h printer)

Returns: BH = state (00h off, 01h on)

AH 22h appears to be some sort of time calculation

AL = 00h unknown

nonzero unknown

AH 23h appears to be some sort of time calculation

AH 24h compare filenames

Call with:

DS:SI pointer to first ASCIZ filename

ES:DI pointer to second ASCIZ filename

Returns:

ZF set if files are same ignoring case and / vs \

AH 25h normalize filename

Call with:

DS:SI pointer to ASCIZ filename

ES:DI pointer to buffer for result

Returns:

filename uppercased, forward slashes changed to backslashes

AH 26h get DOS stack

Returns:

DS:SI pointer to top of character I/O function stack

CX size of stack in bytes

AH 27h increment InDOS flag

AH 28h decrement InDOS flag

Note:
IFS drivers which do not wish to implement functions 20h or 24h-28h may pass the call on to the default handler pointed at by [List-of-Lists+37h].

Format of IFS request block

Offset	Size	Description
00h	WORD	total size in bytes of request
02h	BYTE	class of request
		02h unknown
		03h redirection
		04h unknown
		05h file access
		06h convert error code to string
		07h unknown
03h	WORD	returned DOS error code
05h	BYTE	IFS driver exit status
		00h success
		01h to 04h unknown
		FFh internal failure
06h	*16 BYTEs*	*unknown*
---request class 02h		
16h	BYTE	function code
		04h unknown
17h	*BYTE*	*apparently unused*
18h	*DWORD*	*unknown pointer*
1Ch	DWORD	unknown pointer
20h	*2 BYTEs*	*unknown*
---request class 03h		
16h	BYTE	function code
17h	*BYTE*	*unknown*
18h	*DWORD*	*unknown pointer*
1Ch	*DWORD*	*unknown pointer*
22h	*WORD*	*unknown returned value*
24h	*WORD*	*unknown returned value*
26h	*WORD*	*unknown returned value*
28h	*BYTE*	*unknown returned value*
29h	*BYTE*	*apparently unused*
---request class 04h		
16h	*DWORD*	*unknown pointer*

1Ah	*DWORD*	*unknown pointer*

---request class 05h

16h	BYTE	function code
		01h flush disk space
		02h get disk space
		03h MKDIR
		04h RMDIR
		05h CHDIR
		06h delete file
		07h rename file
		08h search directory
		09h file open/create
		0Ah LSEEK
		0Bh read from file
		0Ch write to file
		0Dh lock region of file
		0Eh commit/close file
		0Fh get/set file attributes
		10h printer control
		11h unknown
		12h process termination
		13h unknown

---class 05h function 01h

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	*4 BYTEs*	*unknown*
26h	*BYTE*	*unknown*
27h	*BYTE*	*unknown*

---class 05h function 02h

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	*4 BYTEs*	*unknown*
26h	WORD	returned total clusters
28h	WORD	returned sectors per cluster
2Ah	WORD	returned bytes per sector
2Ch	WORD	returned available clusters
2Eh	*BYTE*	*unknown returned value*
2Fh	*BYTE*	*unknown*

---class 05h functions 03h, 04h, 05h

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	*4 BYTEs*	*unknown*
26h	DWORD	pointer to directory name

---class 05h function 06h

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	*4 BYTES*	*unknown*
26h	WORD	attribute mask
28h	DWORD	pointer to filename

---class 05h function 07h

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	*4 BYTES*	*unknown*
26h	WORD	attribute mask
28h	DWORD	pointer to source filespec
2Ch	WORD	pointer to destination filespec

---class 05h function 08h

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	*4 BYTEs*	*unknown*
26h	BYTE	00h FINDFIRST
		01h FINDNEXT
28h	DWORD	pointer to FindFIrst search data + 01h if FINDNEXT
2Ch	WORD	search attribute if FINDFIRST
2Eh	DWORD	pointer to filespec if FINDFIRST

---class 05h function 09h

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	DWORD	pointer to IFS open file structure (see below)
26h	*WORD*	*unknown \ together, these specify open vs. create and whether or*
28h	*WORD*	*unknown / not to truncate the file*
2Ah	*4 BYTEs*	*unknown*
2Eh	DWORD	pointer to filename
32h	*4 BYTEs*	*unknown*
36h	WORD	file attributes on call
		unknown returned value
38h	WORD	*unknown returned value*

---class 05h funciton 0Ah

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	DWORD	pointer to IFS open file structure (see below)
26h	BYTE	seek type (02h = from end)
28h	DWORD	offset on call
		returned new absolute position

---class 05h functions 0Bh, 0Ch

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	DWORD	pointer to IFS open file structure (see below)
28h	WORD	number of bytes to transfer
		returned bytes actually transferred
2Ah	DWORD	transfer address

---class 05h function 0Dh

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	DWORD	pointer to IFS open file structure (see below)
26h	*BYTE*	*file handle*
27h	*BYTE*	*apparently unused*
28h	*WORD*	*unknown*
2Ch	*WORD*	*unknown*
2Eh	*WORD*	*unknown*

---class 05h function 0Eh

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	DWORD	pointer to IFS open file structure (see below)
26h	BYTE	00h commit file
		01h close file
27h	*BYTE*	*apparently unused*

---class 05h function 0Fh

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	*4 BYTEs*	*unknown*
26h	BYTE	02h GET attributes
		03h PUT attributes
27h	*BYTE*	*apparently unused*
28h	*12 BYTEs*	*unknown*
34h	*WORD*	*search attributes*

36h	DWORD	pointer to filename
3Ah	*WORD*	*(GET) unknown returned value*
3Ch	*WORD*	*(GET) unknown returned value*
3Eh	*WORD*	*(GET) unknown returned value*
40h	*WORD*	*(GET) unknown returned value*
42h	WORD	(PUT) new attributes
		(GET) returned attributes

---class 05h function 10h

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	DWORD	pointer to IFS open file structure (see below)
26h	*WORD*	*unknown*
28h	*DWORD*	*unknown pointer*
2Ch	*WORD*	*unknown*
2Eh	*BYTE*	*unknown*
2Fh	BYTE	subfunction
		01h get printer setup
		03h to 07h unknown
		21h set printer setup

---class 05h function 11h

17h	*7 BYTEs*	*unknown*
1Eh	*DWORD*	*unknown pointer*
22h	DWORD	pointer to IFS open file structure (see below)
26h	BYTE	subfunction
27h	*BYTE*	*apparently unused*
28h	*WORD*	*unknown*
2Ah	*WORD*	*unknown*
2Ch	*WORD*	*unknown*
2Eh	*BYTE*	*unknown*
2Fh	*BYTE*	*unknown*

---class 05h function 12h

17h	*15 BYTEs*	*apparently unused*
26h	WORD	PSP segment
28h	BYTE	type of process termination
29h	*BYTE*	*apparently unused*

---class 05h function 13h

17h	*15 BYTEs*	*apparently unused*
26h	WORD	PSP segment

---request class 06h

16h	WORD	returned pointer to string corresponding to error code at 03h
1Ah	*BYTE*	*unknown returned value*
1Bh	BYTE	unused

---request class 07h

16h	DWORD	pointer to IFS open file structure (see below)
1Ah	*BYTE*	*unknown*
1Bh	*BYTE*	*apparently unused*

Format of IFS open file structure:

Offset	Size	Description
00h	*WORD*	*unknown*
02h	WORD	device info word
04h	WORD	file open mode
06h	*WORD*	*unknown*
08h	WORD	file attributes
0Ah	WORD	owner's network machine number
0Ch	WORD	owner's PSP segment
0Eh	DWORD	file size
12h	DWORD	current offset in file
16h	WORD	file time
18h	WORD	file date
1Ah	11 BYTEs	filename in FCB format
25h	*WORD*	*unknown*
27h	WORD	hash value of SFT address (low word of linear address + segment & F000h)
29h	3 WORDs	network info from SFT
2Fh	*WORD*	*unknown*

Format of one item in DOS 4 list of special program names:

Offset	Size	Description
00h	BYTE	length of name (00h=end of list)
01h	N BYTEs	name in format name.ext
N+1	*3 BYTEs*	*unknown*

INT 21h Function 53h DOS 2+

TRANSLATE BIOS PARAMETER BLOCK

Compute the information in a Drive Parameter Block from the information in the given BIOS Parameter Block.

Call with:

AH	53h
DS:SI	pointer to BIOS Parameter Block
ES:BP	pointer to buffer for DOS Drive Parameter Block

Returns:

ES:BP buffer filled with a DPB

Format of BIOS Parameter Block:

Offset	Size	Description
00h	WORD	Number of bytes/sector
02h	BYTE	Number of sectors/cluster. Corresponds to (DPB byte 04h) + 1.
03h	WORD	Number of reserved sectors
05h	BYTE	Number of FATs
06h	WORD	Number of root directory entries
08h	WORD	Total number of sectors. Corresponds to: ((DPB bytes 0Dh-0Eh) - 1) * (sectors/cluster) + (DPB Bytes 0Bh-0Ch) For DOS 4.0, set to zero if partition >32M, then set DWORD at 15h to actual number of sectors
0Ah	BYTE	Media descriptor byte
0Bh	WORD	Number of sectors per FAT
DOS 3+		
0Dh	WORD	Number of sectors per track
0Fh	WORD	Number of heads
11h	DWORD	Number of hidden sectors
15h	11 BYTEs	Reserved
DOS 4.0		
15h	DWORD	Total number of sectors if word at 08h contains zero.
19h	6 BYTEs	reserved
1Fh	WORD	Number of cylinders.

21h	BYTE	Device type.
22h	WORD	Device attributes (removable media, etc).

See Also: INT 21/AH=32h

INT 21h Function 55h DOS 2+

CREATE PROGRAM SEGMENT PREFIX

Create a child Program Segment Prefix with the specified amount of available memory, and place it at a given location.

Call with:

AH	55h
DX	segment number at which to set up PSP (see INT 21/AH=26h)
SI	(DOS 3+) value to place in memory size field at DX:[0002h]

Notes:

- This function is like INT 21/AH=26h, but it also sets the memory size to an explicit value rather than copying it from the current PSP and increments the reference count for all inherited files.
- The current PSP segment is set to the segment of the new PSP under DOS 2+.
- Files opened with the "no inherit" flag set are marked as closed in the child PSP.

See Also: INT 21/AH=26h

INT 21h Function 56h DOS 3.1+

RENAME A FILE

Although documented, this call has the undocumented behavior of allowing wildcards in both source and destination when invoked via INT 21/AX=5D00h.

Call with:

AH	56h
DS:DX	pointer to ASCIZ old filespec
ES:DI	pointer to ASCIZ new filespec

Returns:

CF set on error

AX error code (02h,03h,05h,11h) (see *INT 21/AH=59h*)

Notes:

- This function allows moves between directories on the same logical drive.
- When invoked via INT 21/AX=5D00h, error 12h (no more files) is returned on success, and both source and destination specifications must be canonical (as returned by INT 21/AH=60h). Wildcards in the destination are replaced by the corresponding character of each source file being renamed.

See Also: *INT 21/AH=17h*, INT 21/AX=5D00h, INT 21/AH=60h

INT 21h Function 5702h DOS 4.0

GET UNKNOWN INFORMATION

The purpose of this function is not yet known.

Call with:

AX	5702h
BX	*unknown (0000h through 0004h)*
DS:SI	*unknown pointer*
ES:DI	pointer to result buffer
CX	size of result buffer

Returns:

CX size of returned data

INT 21h Function 5703h DOS 4.0

GET UNKNOWN INFORMATION

The purpose of this function is not yet known.

Call with:

AX	5703h
BX	*unknown (0000h through 0004h)*

DS:SI	*unknown pointer*
ES:DI	pointer to result buffer
CX	size of result buffer

Returns:

CX	size of returned data

INT 21h Function 5704h DOS 4.0

UNKNOWN

The purpose of this function is not yet known.

Call with:

AX	5704h
BX	*file handle*
DS:SI	*unknown pointer*
ES:DI	pointer to result buffer
CX	size of result buffer

Returns:

apparently nothing

INT 21h Function 58h DOS 3.0+

GET OR SET ALLOCATION STRATEGY

While sometimes documented (for example, in Microsoft's *MS-DOS Encyclopedia*), this function is included here because some key references (such as IBM's *Technical Reference* for DOS 3.3) do not document it. This function gets or sets the current MS-DOS strategy for allocating memory blocks.

Call with:

AH 58h
AL function code
00h get allocation strategy
01h set allocation strategy
BL strategy code
00h first fit (use first large-enough block)

01h best fit (use smallest large-enough block)
02h last fit (use high part of last usable block)

Returns:

CF set on error
AX error code (01h) (see INT 21h Function 59h)
CF clear if successful
AX strategy code

Note:

The Set subfunction accepts any value in BL; 2 or greater indicates last fit. The Get subfunction returns the last value set, so programs should check whether the value is greater than or equal to 2, not just equal to 2.

INT 21h Function 5D00h DOS 3.1+

SERVER FUNCTION CALL

Execute a specified INT 21h call using the sharing rules for the specified network machine number and process ID.

Call with:

AX 5D00h
DS:DX pointer to DOS parameter list (see below)
DPL contains all register values for a call to INT 21h

Returns:

as appropriate for function being called

Notes:

- This call does not check the requested value in AH. Out of range values will crash the system.
- Sharing retry delay loops are skipped.
- A special sharing mode is enabled.
- Functions which take filenames require canonical names (as returned by INT 21/AH=60h); this is apparently to prevent multi-hop file forwarding.
- Rename (INT 21/AH=56h) and Delete (INT 21/AH=41h) allow wildcards.

Format of DOS parameter list:

Offset	Size	Description
00h	WORD	AX
02h	WORD	BX
04h	WORD	CX
06h	WORD	DX
08h	WORD	SI
0Ah	WORD	DI
0Ch	WORD	DS
0Eh	WORD	ES
10h	WORD	reserved (0)
12h	WORD	computer ID (0000h=current system)
14h	WORD	process ID (PSP segment on specified computer)

See Also: INT 21/AH=60h

INT 21h Function 5D01h DOS 3.1+

COMMIT ALL FILES

Flush all disk buffers and update the directory entry for each file which has been written to since opening or the last commit.

Call with:

AX	5D01h
DS:DX	pointer to DOS parameter list (see INT 21/AX=5D00h), only computer ID and process ID fields used

Returns:

CF set on error

AX error code (see *INT 21/AH=59h*)

Note:

The computer and process IDs are stored but ignored under DOS 3.3

See Also: *INT 21/AH=68h,* INT 2F/AX=1107h

INT 21h Function 5D02h DOS 3.1+

SHARE.EXE - CLOSE FILE BY NAME

Close a file given its fully-qualified name.

Call with:

AX 5D02h
DS:DX pointer to DOS parameter list (see INT 21/AX=5D00h), only fields DX, DS, computer ID, and process ID used
DPL's DS:DX pointer to ASCIZ name of file to close

Returns:

CF set on error
AX error code (see *INT 21/AH=59h*)
CF clear if successful

Notes:

- An error is returned unless SHARE is loaded.
- The name must be a canonical fully-qualified name such as returned by INT 21/AH=60h

See Also: INT 21/AH=52h, INT 21/AX=5D03h, INT 21/AX=5D04h, *INT 21/AH=3Eh*, INT 21/AH=60h

INT 21h Function 5D03h DOS 3.1+

SHARE.EXE - CLOSE ALL FILES FOR GIVEN COMPUTER

Close all files which were opened using a particular network machine number.

Call with:

AX 5D03h
DS:DX pointer to DOS parameter list (see INT 21/AX=5D00h), only computer ID used

Returns:

CF set on error
AX error code (see *INT 21/AH=59h*)
CF clear if successful

Note:
An error is returned unless SHARE is loaded.

See Also: INT 21/AH=52h, INT 21/AX=5D02h, INT 21/AX=5D04h

INT 21h Function 5D04h DOS 3.1+

SHARE.EXE - CLOSE ALL FILES FOR GIVEN PROCESS

Close all files which were opened by a particular process.

Call with:

AX	5D04h
DS:DX	pointer to DOS parameter list (see INT 21/AX=5D00h), only computer ID and process ID fields used

Returns:
CF set on error
AX error code (see *INT 21/AH=59h*)
CF clear if successful

Note:
An error is returned unless SHARE is loaded.

See Also: INT 21/AH=52h, INT 21/AX=5D02h, INT 21/AX=5D03h

INT 21h Function 5D04h DOS 3.1+

SHARE.EXE - GET OPEN FILE LIST ENTRY

Return the filename and some additional information about a specified entry in SHARE's internal data structures.

Call with:

AX	5D05h
DS:DX	pointer to DOS parameter list (see INT 21/AX=5D00h)
DPL's BX	index of sharing record
DPL's CX	index of SFT in sharing record's SFT list

Returns:

CF clear if successful

ES:DI	pointer to ASCIZ filename
BX	network machine number of SFT's owner
CX	number of locks held by SFT's owner

CF set if either index out of range

AX	0012h (no more files)

Notes:

- An error is returned unless SHARE is loaded.
- The returned filenames are be canonical fully-qualified names such as returned by INT 21/AH=60h

See Also: INT 21/AH=52h, INT 21/AH=5Ch, INT 21/AH=60h

INT 21h Function 5D06h DOS 3.0+

GET ADDRESS OF DOS SWAPPABLE DATA AREA

Return the address and size of the region which must be swapped out and restored to allow DOS to be reentered.

Call with:

AX 5D06h

Returns:

CF set on error

AX	error code (see *INT 21/AH=59h*)

CF clear if successful

DS:SI	pointer to nonreentrant data area (includes all three DOS stacks) (critical error flag is first byte)
CX	size in bytes of area which must be swapped while in DOS
DX	size in bytes of area which must always be swapped

Notes:

- The Critical Error flag is used in conjunction with the InDOS flag (see INT 21/AH=34h) to determine when it is safe to enter DOS from a TSR.

- Setting the CritErr flag allows use of functions 50h/51h from INT 28h under DOS 2.x by forcing use of the correct stack.
- Swapping the data area allows DOS to be reentered unless DOS is in a critical section delimited by INT 2A/AH=80h and INT 2A/AH=81h,82h.
- Under DOS 4.0, INT 21/AX=5D0Bh should be used instead of this function.

Format of DOS 3.10-3.30 Swappable Data Area:

Offset	Size	Description
00h	BYTE	critical error flag
01h	BYTE	InDOS flag (count of active INT 21h calls)
02h	*BYTE*	*drive on which current critical error occurred or FFh*
03h	BYTE	locus of last error
04h	WORD	extended error code of last error
06h	BYTE	suggested action for last error
07h	BYTE	class of last error
08h	DWORD	ES:DI pointer for last error
0Ch	DWORD	current DTA
10h	WORD	current PSP
12h	WORD	stores SP across an INT 23
14h	WORD	return code from last process termination (cleared after reading with AH=4Dh)
16h	BYTE	current drive
17h	BYTE	extended break flag
remainder need only be swapped if in DOS		
18h	WORD	value of AX on call to INT 21
1Ah	WORD	PSP segment for sharing/network
1Ch	WORD	network machine number for sharing/network (0000h=current system)
1Eh	WORD	first usable memory block found when allocating memory
20h	WORD	best usable memory block found when allocating memory
22h	WORD	last usable memory block found when allocating memory
24h	*2 BYTEs*	*apparently not referenced by kernel*
26h	*WORD*	*unknown*
28h	*BYTE*	*unknown*
29h	*BYTE*	*unknown*
2Ah	*BYTE*	*unknown*
2Bh	*BYTE*	*flag of some kind*
2Ch	*BYTE*	*flag of some kind*
2Dh	*BYTE*	*apparently not referenced by kernel*

2Eh	BYTE	day of month
2Fh	BYTE	month
30h	WORD	year - 1980
32h	WORD	number of days since 1-1-1980
32h	BYTE	day of week (0=Sunday)
35h	*BYTE*	*unknown*
36h	*BYTE*	*unknown flag*
37h	*BYTE*	*unknown flag*
38h	26 BYTEs	device driver request header
52h	DWORD	pointer to device driver entry point (used in calling driver)
56h	22 BYTEs	device driver request header
6Ch	22 BYTEs	device driver request header
82h	BYTE	type of PSP copy (00h=simple copy for INT 21/AH=26h, FFh=make child)
83h	*BYTE*	*apparently not referenced by kernel*
84h	*WORD*	*unknown*
86h	*WORD*	*unknown*
88h	*2 BYTEs*	*unknown*
8Ah	6 BYTEs	CLOCK$ transfer record (see INT 21/AH=52h)
90h	*2 BYTEs*	*unknown*
92h	128 BYTEs	buffer for filename
112h	128 BYTEs	buffer for filename
192h	21 BYTEs	findfirst/findnext search data block (see INT 21/AH=4Eh)
1A7h	32 BYTEs	directory entry for found file
1C7h	81 BYTEs	copy of current directory structure for drive being accessed
218h	*11 BYTEs*	*FCB-format filename, use unknown*
223h	*BYTE*	*unknown*
224h	11 BYTEs	wildcard destination specification for rename (FCB format)
22Fh	*2 BYTEs*	*unknown*
231h	*WORD*	*unknown*
233h	*5 BYTEs*	*unknown*
238h	BYTE	directory search attributes
239h	BYTE	type of FCB (00h regular, FFh extended)
23Ah	BYTE	extended FCB file attribute (find first search attr mask)
23Bh	*BYTE*	*file open mode*
23Ch	*BYTE*	*unknown flag bits*
23Dh	*BYTE*	*unknown flag or counter*
23Eh	*BYTE*	*unknown flag*
23Fh	BYTE	flag indicating how DOS function was invoked (00h if direct INT 20/INT 21, FFh if server call AX=5D00h)

240h	*WORD*	*unknown*
242h	*BYTE*	*unknown*
243h	*BYTE*	*unknown*
244h	*BYTE*	*unknown*
245h	*BYTE*	*unknown flag or counter*
246h	*BYTE*	*unknown flag*
247h	*BYTE*	*unknown flag*
248h	*BYTE*	*unknown flag*
249h	BYTE	type of process termination (00h-03h)
24Ah	*BYTE*	*unknown flag*
24Bh	BYTE	value with which to replace the first byte of a deleted file's name (normally E5h, but set to 00h as described under INT 21/AH=13h)
24Ch	DWORD	pointer to Drive Parameter Block for critical error invocation
250h	DWORD	pointer to stack frame containing user registers on INT 21
254h	*WORD*	*stores SP*
256h	*DWORD*	*pointer to DOS Drive Parameter Block for unknown use*
25Ah	*WORD*	*unknown*
25Ch	*WORD*	*unknown*
25Eh	*WORD*	*unknown flag*
260h	*WORD*	*unknown*
262h	BYTE	Media ID byte returned by INT 21/AH=1Bh,1Ch
263h	*BYTE*	*appears not to be referenced by kernel*
264h	*DWORD*	*unknown pointer*
268h	DWORD	pointer to current SFT
26Ch	DWORD	pointer to current directory structure for drive being accessed
270h	DWORD	pointer to caller's FCB
274h	*WORD*	*unknown*
276h	WORD	temporary storage for file handle
278h	DWORD	pointer to a JFT entry in process handle table (see INT 21/AH=26h)
27Ch	WORD	offset in DOS CS of first filename argument
27Eh	WORD	offset in DOS CS of second filename argument
280h	*WORD*	*unknown*
282h	*WORD*	*unknown*
284h	*WORD*	*unknown*
286h	*WORD*	*unknown*
288h	*WORD*	*unknown*
28Ah	*WORD*	*unknown*
28Ch	*WORD*	*unknown*
28Eh	*2 BYTEs*	*unknown*

290h	*WORD*	*unknown*
292h	DWORD	current offset in file
296h	*WORD*	*unknown*
298h	*WORD*	*unknown*
29Ah	*WORD*	*unknown*
29Ch	*WORD*	*unknown*
29Eh	*WORD*	*unknown*
2A0h	*WORD*	*unknown*
2A2h	DWORD	number of bytes appended to file
2A6h	*DWORD*	*pointer to disk buffer*
2AAh	*DWORD*	*pointer to a System File Table*
2AEh	WORD	used by INT 21h dispatcher to store caller's BX
2B0h	WORD	used by INT 21h dispatcher to store caller's DS
2B2h	WORD	temporary storage while saving/restoring caller's registers
2B4h	DWORD	pointer to prev call frame (offset 250h) if INT 21h reentered also switched to for duration of INT 24h
2B8h	21 BYTEs	FindFirst search data for source file(s) of a rename operation (see INT 21/AH=4Eh)
2CDh	32 BYTEs	directory entry for file being renamed
2EDh	331 BYTEs	critical error stack
438h	384 BYTEs	disk stack (functions greater than 0Ch, INT 25, INT 26)
5B8h	384 BYTEs	character I/O stack (functions 01h through 0Ch)

DOS 3.3 only

738h	BYTE	flag affecting AH=08h (see INT 21/AH=64h)
739h	*BYTE*	*unknown, appears to be a drive number*
73Ah	*BYTE*	*unknown flag*
73Ah	*BYTE*	*unknown*

See Also: INT 21/AX=5D0Bh, INT 2A/AH=80h, INT 2A/AH=81h, INT 2A/AH=82h

INT 21h Function 5D07h — DOS 3.1+ network

GET REDIRECTED PRINTER MODE

Determine whether redirected printer output is treated as a single print job or as multiple print jobs.

Call with:

AX	5D07h

Returns:

DL mode
- 00h redirected output is combined
- 01h redirected output in separate print jobs

See Also: INT 21/AX=5D08h, INT 21/AX=5D09h, INT 2F/AX=1125h

INT 21h Function 5D08h — DOS 3.1+ network

SET REDIRECTED PRINTER MODE

Specify whether redirected printer output should be treated as a single print job or as multiple print jobs.

Call with:

AX 5D08h

DL mode
- 00h redirected output is combined
- 01h redirected output placed in separate jobs, start new print job now

See Also: INT 21/AX=5D07h, INT 21/AX=5D09h, INT 2F/AX=1125h

INT 21h Function 5D09h — DOS 3.1+ network

FLUSH REDIRECTED PRINTER OUTPUT

Force all redirected printer output to be sent to the printer, and start a new print job.

Call with:

AH 5D09h

See Also: INT 21/AX=5D07h, INT 21/AX=5D08h, INT 2F/AX=1125h

INT 21h Function 5D0Ah DOS 3.1+

SET EXTENDED ERROR INFORMATION

Set the values to be returned by the next "Get Extended Error Code" call.

Call with:

AX	5D0Ah
DS:DX	pointer to DOS Parameter List (see INT 21/AX=5D00h)

Returns:

Nothing. The next call to AH=59h will return the values from fields AX, BX, CX, DX, DI, and ES in the corresponding registers.

See Also: *INT 21/AH=59h*

INT 21h Function 5D0Bh DOS 4.x

GET DOS SWAPPABLE DATA AREAS

Return the address of a list of regions which must be swapped out and restored to allow DOS to be reentered.

Call with:

AX	5D0Bh

Returns:

CF set on error

AX	error code (see *INT 21/AH=59h*)

CF clear if successful

DS:SI	pointer to swappable data area list (see below)

Note:

Copying and restoring the swappable data areas allows DOS to be reentered unless it is in a critical section delimited by calls to INT 2A/AH=80h and INT 2A/AH=81h or AH=82h.

Format of swappable data area list:

Offset	Size	Description
00h	WORD	count of data areas

02h	N BYTEs	"count" copies of data area record	
	Offset	Size	Description
	00h	DWORD	address
	04h	WORD	length and type
			bit 15 set if swap always, clear if swap in DOS
			bits 14-0: length in bytes

Format of PCDOS 4.01 swappable data area:

Offset	Size	Description
00h	BYTE	critical error flag
01h	BYTE	InDOS flag (count of active INT 21h calls)
02h	*BYTE*	*drive on which current critical error occurred or FFh*
03h	BYTE	locus of last error
04h	WORD	extended error code of last error
06h	BYTE	suggested action for last error
07h	BYTE	class of last error
08h	DWORD	ES:DI pointer for last error
0Ch	DWORD	current DTA
10h	WORD	current PSP
12h	WORD	stores SP across an INT 23
14h	WORD	return code from last process termination (cleared after reading with AH=4Dh)
16h	BYTE	current drive
17h	BYTE	extended break flag
18h	*2 BYTEs*	*unknown*
remainder need only be swapped if in DOS		
1Ah	WORD	value of AX on call to INT 21
1Ch	WORD	PSP segment for sharing/network
1Eh	WORD	network machine number for sharing/network (0000h=current system)
20h	WORD	first usable memory block found when allocating memory
22h	WORD	best usable memory block found when allocating memory
24h	WORD	last usable memory block found when allocating memory
26h	*2 BYTEs*	*apparently not referenced by kernel*
28h	*WORD*	*unknown*
2Ah	*BYTE*	*unknown*
2Bh	*BYTE*	*unknown*
2Ch	*BYTE*	*unknown*
2Dh	*BYTE*	*unknown*
2Eh	*BYTE*	*unknown*

2Fh	*BYTE*	*apparently not referenced by kernel*
30h	BYTE	day of month
31h	BYTE	month
32h	WORD	year - 1980
34h	WORD	number of days since 1-1-1980
36h	BYTE	day of week (0=Sunday)
37h	*BYTE*	*unknown*
38h	*BYTE*	*unknown*
39h	*BYTE*	*unknown*
38h	30 BYTEs	device driver request header
58h	DWORD	pointer to device driver entry point (used in calling driver)
5Ch	22 BYTEs	device driver request header
72h	30 BYTEs	device driver request header
90h	*6 BYTEs*	*unknown*
96h	6 BYTEs	CLOCK$ transfer record (see INT 21/AH=52h)
9Ch	*2 BYTEs*	*unknown*
9Eh	128 BYTEs	buffer for filename
11Eh	128 BYTEs	buffer for filename
19Eh	21 BYTEs	findfirst/findnext search data block (see INT 21/AH=4Eh)
1B3h	32 BYTEs	directory entry for found file
1D3h	88 BYTEs	copy of current directory structure for drive being accessed
22Bh	11 BYTEs	FCB-format filename, use unknown
236h	*BYTE*	*unknown*
237h	11 BYTEs	wildcard destination specification for rename (FCB format)
242h	*2 BYTEs*	*unknown*
244h	*WORD*	*unknown*
246h	*5 BYTEs*	*unknown*
24Bh	*BYTE*	*unknown*
24Ch	*BYTE*	*unknown*
24Dh	*BYTE*	*attribute mask for directory search*
24Eh	*BYTE*	*unknown*
24Fh	*BYTE*	*unknown flag bits*
250h	*BYTE*	*unknown*
251h	*BYTE*	*unknown*
252h	BYTE	flag indicating how DOS function was invoked (00h if direct INT 20/INT 21, FFh if server call AX=5D00h)
253h	*BYTE*	*unknown*
254h	*BYTE*	*unknown*
255h	*BYTE*	*unknown*

256h	*BYTE*	*unknown*
257h	*BYTE*	*unknown*
258h	*BYTE*	*unknown*
259h	*BYTE*	*unknown*
25Ah	*BYTE*	*unknown*
25Bh	*BYTE*	*unknown*
25Ch	BYTE	type of process termination (00h-03h)
25Dh	*BYTE*	*unknown*
25Eh	*BYTE*	*unknown*
25Fh	*BYTE*	*unknown*
260h	DWORD	pointer to Drive Parameter Block for critical error invocation
264h	DWORD	pointer to stack frame containing user registers on INT 21
268h	*WORD*	*stores SP*
26Ah	*DWORD*	*pointer to DOS Drive Parameter Block for unknown use*
26Eh	WORD	segment of disk buffer
270h	*WORD*	*unknown*
272h	*WORD*	*unknown*
274h	*WORD*	*unknown*
276h	*WORD*	*unknown*
278h	BYTE	Media ID byte returned by INT 21/AH=1Bh,1Ch
279h	*BYTE*	*apparently not referenced by kernel*
27Ah	*DWORD*	*unknown pointer*
27Eh	DWORD	pointer to current SFT
282h	DWORD	pointer to current directory structure for drive being accessed
286h	DWORD	pointer to caller's FCB
28Ah	*WORD*	*unknown*
28Ch	*WORD*	*unknown*
28Eh	DWORD	pointer to a JFT entry in process handle table (see INT 21/AH=26h)
292h	WORD	offset in DOS CS of first filename argument
294h	WORD	offset in DOS CS of second filename argument
296h	*WORD*	*unknown*
298h	*WORD*	*unknown*
29Ah	*WORD*	*unknown*
29Ch	*WORD*	*unknown*
29Eh	*WORD*	*unknown*
2A0h	*WORD*	*unknown*
2A2h	*WORD*	*appears to be directory cluster number*
2A4h	*DWORD*	*unknown*
2A8h	*DWORD*	*unknown*

2ACh	*WORD*	*unknown*
2AEh	*DWORD*	*offset in file*
2B2h	*WORD*	*unknown*
2B4h	WORD	bytes in partial sector
2B6h	WORD	number of sectors
2B8h	*WORD*	*unknown*
2BAh	*WORD*	*unknown*
2BCh	*WORD*	*unknown*
2BEh	DWORD	number of bytes appended to file
2C2h	*DWORD*	*pointer to a disk buffer for unknown use*
2C6h	*DWORD*	*pointer to a System File Table for unknown use*
2CAh	WORD	used by INT 21h dispatcher to store caller's BX
2CCh	WORD	used by INT 21h dispatcher to store caller's DS
2CEh	WORD	temporary storage while saving/restoring caller's registers
2D0h	DWORD	pointer to prev call frame (offset 264h) if INT 21h reentered also switched to for duration of INT 24
2D4h	*WORD*	*unknown*
2D6h	*BYTE*	*unknown*
2D7h	*WORD*	*unknown*
2D9h	*DWORD*	*unknown pointer*
2DDh	*WORD*	*unknown*
2DFh	*WORD*	*unknown*
2E1h	*WORD*	*unknown*
2E3h	*DWORD*	*unknown*
2E7h	*WORD*	*unknown*
2E9h	*WORD*	*unknown*
2EBh	*BYTE*	*unknown*
2ECh	WORD	stores DS during call to [List-of-Lists + 37h]
2EEh	*WORD*	*unknown*
2F0h	*BYTE*	*unknown*
2F1h	*WORD*	*unknown bit flags*
2F3h	DWORD	pointer to user-supplied filename
2F7h	*DWORD*	*unknown pointer*
2FBh	WORD	stores SS during call to [List-of-Lists + 37h]
2FDh	WORD	stores SP during call to [List-of-Lists + 37h]
2FFh	BYTE	flag, nonzero if stack switched in calling [List-of-Lists+37h]
300h	21 BYTEs	FindFirst search data for source file(s) of a rename operation (see INT 21/AH=4Eh)
315h	32 BYTEs	directory entry for file being renamed

335h	331 BYTEs	critical error stack
480h	384 BYTEs	disk stack (functions greater than 0Ch, INT 25, INT 26)
600h	384 BYTEs	character I/O stack (functions 01h through 0Ch)
780h	BYTE	flag affecting AH=08h (see INT 21/AH=64h)
781h	*BYTE*	*appears to be a drive number*
782h	*BYTE*	*unknown flag*
783h	*BYTE*	*unknown*
784h	*WORD*	*unknown*
786h	*WORD*	*unknown*
788h	*WORD*	*unknown*
78Ah	*WORD*	*unknown*

See Also: INT 21/AX=5D06h, INT 2A/AH=80h, INT 2A/AH=81h, INT 2A/AH=82h

INT 21h Function 5E01h DOS 3.1+ network

SET MACHINE NAME

Specify the system's network machine name and number.

Call with:

AX	5E01h	
CH	00h	undefine name
	other	define name
CL	name number	
DS:DX	pointer to 15-character blank-padded ASCIZ name	

See Also: *INT 21/AX=5E00h*

INT 21h Function 5E04h DOS 3.1+ network

SET PRINTER MODE

Specify whether the printer should be operated in text or binary mode.

Call with:

AX	5E04h
BX	redirection list index

DX mode
bit 0: set if binary, clear if text (tabs expanded to blanks)

Returns:
CF set on error
AX error code (see *INT 21/AH=59h*)

Note:
Calls INT 2F/AX=111Fh with 5E04h on stack.

See Also: INT 21/AX=5E05h, INT 2F/AX=111Fh

INT 21h Function 5E05h DOS 3.1+ network

GET PRINTER MODE

Determine whether the printer is being operated in text or binary mode.

Call with:
AX 5E05h
BX redirection list index

Returns:
CF set on error
AX error code (see *INT 21/AH=59h*)
CF clear if successful
DX printer mode (see *INT 21/AX=5E04h*)

Note:
Calls INT 2F/AX=111Fh with 5E05h on stack.

See Also: INT 21/AX=5E04h, INT 2F/AX=111Fh

INT 21h Function 5F00h DOS 3.1+ network

GET REDIRECTION MODE

Determine whether disk or printer redirection is current enabled.

Call with:

AX 5F00h
BL redirection type
03h printer
04h disk drive

Returns:

CF set on error
AX error code (see *INT 21/AH=59h*)
CF clear if successful
BH redirection state
00h off
01h on

See Also: INT 21/AX=5F01h

INT 21h Function 5F01h DOS 3.1+ network

SET REDIRECTION MODE

Specify whether disk or printer redirection is to be enabled or disabled.

Call with:

AX 5F01h
BL redirection type
03h printer
04h disk drive
BH redirection state
00h off
01h on

Returns:

CF set on error
AX error code (see *INT 21/AH=59h*)

Note:
When redirection is off, the local device (if any) rather than the remote device is used.

See Also: INT 21/AX=5F00h, INT 2F/AX=111Eh

INT 21h Function 5F05h DOS 4+ network

GET REDIRECTION LIST EXTENDED ENTRY

Return the source and target of a given redirection, as well as its status and type.

Call with:

AX	5F05h
BX	redirection list index
DS:SI	pointer to buffer for ASCIZ source device name
ES:DI	pointer to buffer for destination ASCIZ network path

Returns:

CF set on error
- AX error code (see *INT 21/AH=59h*)

CF clear if successful
- BH device status flag (bit 0 clear if valid)
- BL device type (03h if printer, 04h if drive)
- CX stored parameter value (user data)
- BP NETBIOS local session number
- DS:SI buffer filled
- ES:DI buffer filled

Note:
The local session number allows sharing the redirector's session number. However, if an error is caused on the NETBIOS LSN, the redirector may be unable to correctly recover from subsequent errors.

See Also: INT 2F/AX=111Eh

INT 21h Function 5F06h DOS 4+ network

GET REDIRECTION LIST

This function appears to be similar to INT 21/AX=5F02h (get redirection list) and INT 21/AX=5F05h (get redirection list extended entry).

Call with:

AX 5F06h
additional arguments (if any) unknown

Returns:

unknown

See Also: INT 21/AX=5F05h, INT 2F/AX=111Eh

INT 21h Function 60h DOS 3+ internal

RESOLVE PATH STRING TO CANONICAL PATH STRING

Given a file specification, return an absolute pathname which takes into account any renaming due to JOIN, SUBST, ASSIGN, or network redirections.

Call with:

AH 60h
DS:SI pointer to ASCIZ relative path string or directory name
ES:DI pointer to 128-byte buffer for ASCIZ canonical fully qualified name

Returns:

CF set on error
    AX error code
        02h invalid source name
        03h invalid drive or malformed path
CF clear if successful
    AH 00h
    AL destroyed (00h or 5Ch or last character of current directory on drive)
    buffer filled with qualified name of the form D:\PATH\FILE.EXT or
        \\MACHINE\PATH\FILE.EXT

Notes:

- The input path need not actually exist.
- Letters are converted to uppercase, forward slashes are converted to backslashes, asterisks are converted to the appropriate number of question marks, and file and directory names are truncated to 8.3 characters if necessary. Additionally, '.' and '..' entries in the path are resolved.
- Qualified filespecs on local drives always start with "d:", those on network drives always start with "\\".
- If the given path string is on a JOINed drive, the returned name is the one that would be needed if the drive were not JOINed; similarly for a SUBSTed, ASSIGNed, or network drive letter. Because of this, it is possible to get a qualified name that is not legal under the current combination of SUBSTs, ASSIGNs, JOINs, and network redirections.
- Functions which take pathnames require canonical paths if invoked via the INT 21/AX=5D00h server call mechanism.
- This function is supported by the OS/2 v1.1 compatibility box.

See Also: INT 2F/AX=1123h, INT 2F/AX=1221h

INT 21h Function 61h DOS 3+

UNUSED

This function performs no action and returns immediately.

Call with:

AH 61h

Returns:

AL 00h

INT 21h Function 63h DOS 2.25 only

GET LEAD BYTE TABLE (2-BYTE CHARACTER SUPPORT)

The subfunctions of this call provide additional foreign-language support.

Call with:

AH 63h
AL subfunction
00h get system lead byte table
Returns:
DS:SI pointer to lead byte table
01h set/clear interim console flag (determine whether interim bytes are returned on some console functions)
DL 01h/00h to set/clear interim console flag
02h get interim console flag
Returns:
DL interim console flag

Returns:

CF set on error
AX error code (01h) (see *INT 21/AH=59h*)
CF clear if successful
DS:SI pointer to lead byte table (subfunction 00h only)
DL interim console flag (subfunction 02h only)

Note:

These calls do not preserve any registers other than CS:IP and SS:SP.

INT 21h Function 6300h — Asian DOS 3.2+ only

GET DOUBLE BYTE CHARACTER SET LEAD TABLE

Return a list of the ranges of characters which are the first half of a two-byte character.

Call with:

AX 6300h

Returns:

AL error code
00h successful
DS:SI pointer to DBCS table (see below)
BX, CX, DX, BP, DI, and ES destroyed
FFh not supported

Notes:
- This call is probably identical to INT 21/AX=6300h for DOS 2.25.
- The US version of DOS 4.0 accepts this function, but returns an empty list.

Format of DBCS table:

Offset	Size	Description
00h	2 BYTEs	low/high ends of a range of leading byte of double-byte chars
02h	2 BYTEs	low/high ends of a range of leading byte of double-byte chars
		...
N	2 BYTEs	00h,00h end flag

INT 21h Function 6301h — Asian DOS 3.2+ only

SET KOREAN (HONGEUL) INPUT MODE

Specify whether console input functions are allowed to return partially-formed multi-keystroke characters.

Call with:
AX 6301h
DL new mode
00h return only full characters on DOS keyboard input functions
01h return partially-formed characters also

Returns:
AL status
00h successful
FFh invalid mode

See Also: INT 21/AX=6302h

INT 21h Function 6302h — Asian DOS 3.2+ only

GET KOREAN (HONGEUL) INPUT MODE

Determine whether console input functions will return partially-formed multi-keystroke characters.

Call with:
AX 6302h

Returns:

AL status
- 00h successful
  - DL current input mode
    - 00h return only full characters
    - 01h return partial characters
- FFh not supported

See Also: INT 21/AX=6301h

INT 21h Function 64h — DOS 3.2 only

GET/SET UNKNOWN FLAG

The purpose of this function is not known.

Call with:

AH 64h

AL subfunction
- 00h get value
  - **Returns:**
    - *DL unknown*
- 01h set value
  - *DL unknown*
- 02h get and set value
  - DL new value
  - **Returns:**
    - DL old value

INT 21h Function 64h — DOS 3.3+

SET UNKNOWN FLAG

The purpose of this function is not known, other than the fact that the flag is used only by INT 21/AH=08h.

Call with:

AH 64h

AL	flag	
	00h	*unknown*
	nonzero	*unknown*

Returns:
nothing

Note:
- This function is called by DOS 3.3+ PRINT.COM.
- This function uses caller's stack, and is therefore reentrant.

INT 21h Function 6505h DOS 3.3+

GET POINTER TO FILENAME TERMINATOR TABLE

Return information about the characters which terminate a filename.

Call with:

AX	6505h
BX	code page (-1=global code page)
DX	country ID (-1=current country)
ES:DI	pointer to country information buffer (see below)
CX	size of buffer (>= 5)

Returns:
CF set on error

AX	error code (see *INT 21/AH=59h*)

CF clear if succesful

CX	size of country information returned
ES:DI	pointer to country information

Notes:
- This function appears to return the same information for all countries and code pages.
- NLSFUNC must be installed to get information for countries other than the default.

Format of country information:

Offset	Size	Description
00h	BYTE	info ID
01h	DWORD	pointer to filename character table (see below)

Format of filename terminator table:

Offset	Size	Description
00h	WORD	table size
02h	*7 BYTEs*	*unknown (01h 00h FFh 00h 00h 20h 02h in MSDOS 3.30)*
09h	BYTE	length of following data
0Ah	N BYTES	characters which terminate a filename: <>."/\[]:|+=;,

See Also: INT 21/AH=38h, INT 2F/AX=1401h, INT 2F/AX=1402h

INT 21h Function 65h DOS 4+

COUNTRY-DEPENDENT CHARACTER CAPITALIZATION

Capitalize a character or string using the capitalization rules for the current country.

Call with:

AH 65h
AL function
- 20h capitalize character
  - DL character to capitalize
- 21h capitalize string
  - DS:DX pointer to string to capitalize
  - CX length of string
- 22h capitalize ASCIZ string
  - DS:DX pointer to ASCIZ string to capitalize

Returns:

CF set on error
AX error code (see *INT 21/AH=59h*)
CF clear on success
DL = capitalized character (function 00h only)

INT 21h Function 6523h DOS 4+

DETERMINE IF CHARACTER REPRESENTS YES/NO RESPONSE

Compare the specified character against the YES and NO responses for the current country.

Call with:

AX 6523h
DL character
DH second character of double-byte character (if applicable)

Returns:

CF set on error
CF clear if successful
AX type
00h no
01h yes
02h neither yes nor no

INT 21h Function 65h DOS 4+

COUNTRY-DEPENDENT FILENAME CAPITALIZATION

Capitalize a filename character or string using the filename capitalization rules for the current country.

Call with:

AH 65h
AL function
A0h capitalize filename character
DL character to capitalize
Returns:
DL capitalized character
A1h capitalize counted filename string
DS:DX pointer to filename string to capitalize
CX length of string
A2h capitalize ASCIZ filename
DS:DX pointer to ASCIZ filename to capitalize

Returns:

CF set on error
    AX error code (see *INT 21/AH=59h*)

Note:

These calls are nonfunctional in DOS 4.00 and 4.01 due to a bug.

INT 21 Function 67h — DOS 3.3+

SET HANDLE COUNT

Although documented, this function is included because of a bug in early releases. This function is used to increase the per-process limit on open files beyond the default of 20 files.

Call with:

AH 67h
BX desired number of handles

Returns:

CF set on error
    AX error code (see *INT 21/AH=59h*)
CF clear if successful

Notes:

- No action is taken if BX is <= 20.
- Only the first 20 handles are copied to child processes in DOS 3.3. Although it is perfectly legal to specify more that 255 handles, it is not possible to use more than 255 unless some handles are duplicates created with INT 21/AH=45h or INT 21/AH=46h.

BUG:

The original release of DOS 3.3 allocates a full 64K for the handle table on requests for an even number of handles.

See Also: INT 21/AH=26h

INT 21h Function 69h DOS 4+

GET/SET DISK SERIAL NUMBER

Determine or specify a disk's serial number and volume label.

Call with:

AH	69h
AL	subfunction
	00h get serial number
	01h set serial number
BL	drive (0=default, 1=A, 2=B, etc)
DS:DX	pointer to disk info (see below)

Returns:

CF set on error
    AX error code (see *INT 21/AH=59h*)
CF clear if successful
    AX destroyed
    (AL=00h) buffer filled with appropriate values from extended BPB
    (AL=01h) extended BPB on disk set to values from buffer

Notes:

- This function will not generate a critical error; all errors are returned in AX.
- Error 0005h is returned if there is no extended BPB on the disk.
- This function does not work on network drives, and returns error 0001h if used on a network drive.
- The buffer after the first two bytes is an exact copy of bytes 27h thru 3Dh of the extended BPB on the disk.

Format of disk info:

Offset	Size	Description
00h	WORD	info level (zero)
02h	DWORD	disk serial number (binary)
06h	11 BYTEs	volume label or "NO NAME " if none present
11h	8 BYTEs	(AL=00h only) filesystem type—string "FAT12 " or "FAT16 "

INT 21h Function 6Ah DOS 4+

UNKNOWN

The purpose of this function is not known.

Call with:

AH 6Ah
additional arguments (if any) unknown

Returns:

unknown

INT 21h Function 6Bh DOS 4+

UNKNOWN

The purpose of this function is not known, but it appears to be related to installable file systems.

Call with:

AH 6Bh
AL subfunction
00h unknown
DS:SI pointer to Current Directory Structure
CL drive (1=A:)
01h unknown
DS:SI unknown pointer
CL file handle
02h unknown
DS:SI pointer to Current Directory Structure
DI unknown
CX drive (1=A:)
additional arguments (if any) unknown

Returns:

unknown

Note:

This call is passed directly through to INT 2F/AX=112Fh, with the caller's AX on the top of the stack.

INT 28h DOS 2+

KEYBOARD BUSY LOOP

This interrupt is called from inside the "get input from keyboard" routine in DOS, if and only if it is safe to use INT 21h Functions 0D and higher at that time even if the state of the INDOS flag (see INT 21/AH=34h) indicates otherwise. It is used primarily by the PRINT.COM routines and TSR programs, but any number of other routines could be chained to it by saving the original vector, and calling it (or just JMPing to it) at the end of the new routine.

Notes:
- This interrupt is supported by the OS/2 compatibility box.
- The INT 28h handler may invoke any INT 21h function except functions 00h through 0Ch (and 50h/51h under DOS 2.xx unless the DOS Critical Error flag is set).
- Calls to INT 21/AH=3Fh and INT 21/AH=40h made from inside the INT 28h handler may not use a handle which refers to CON.
- Until a program installs its own routine, this interrupt vector points to an IRET opcode.

See Also: INT 2A/AH=84h

INT 29h DOS 2+

FAST PUTCHAR

This interrupt is called from the DOS output routines when sending characters to a device whose attribute word has bit 4 set.

Call with:
AL character to display

Returns:
nothing

Notes:
- The default handler under DOS 2.x and 3.x simply calls *INT 10/AH=0Eh*.
- The default handler under DESQview 2.2 understands the <ESC>[2J screen-clearing sequence, but calls *INT 10/AH=0Eh* otherwise.
- COMMAND.COM v3.3 compares the vectors for INT 20h and INT 29h, and assumes that ANSI.SYS is installed if the segment of INT 29h is greater than the segment of INT 20h.

INT 2Ah Function 00h network

INSTALLATION CHECK

Determine whether a Microsoft Networks-compatible network is installed.

Call with:

AH 00h

Returns:

AH nonzero if installed

INT 2Ah Function 01h network

EXECUTE NETBIOS REQUEST, NO ERROR RETRY

This call is equivalent to invoking *INT 5C*, the NETBIOS interrupt.

Call with:

AH 01h
ES:BX pointer to NCB (see *INT 5C*)

Returns:

AL NetBIOS error code
AH 00h if no error
01h on error

See Also: INT 2A/AH=04h, *INT 5C*

INT 2Ah Function 02h network

SET NETWORK PRINTER MODE

Call with:

AH 02h
additional arguments (if any) unknown

Returns:

unknown

INT 2Ah Function 03h network

CHECK DIRECT I/O

This function determines whether direct transfers to a disk are allowed.

Call with:

AX 0300h
DS:SI pointer to ASCIZ disk device name (full path or only drive specifier—must include the colon)

Returns:

CF clear if direct disk access allowed
set if disallowed

Notes:

- Do not use direct disk accesses if this function returns CF set or the device is redirected (INT 21/AX=5F02h).
- This function may take some time to execute.

See Also: *INT 13, INT 25, INT 26,* INT 21/AX=5F02h

INT 2Ah Function 04h network

EXECUTE NETBIOS REQUEST

Invoke the NETBIOS handler, optionally retrying the operation on certain errors.

Call with:

AH 04h
AL 00h for error retry, 01h for no retry
ES:BX pointer to NCB (see *INT 5C*)

Returns:

AX 0000h for no error
AH 01h
AL error code

Note:
The request is automatically retried (if AL=00h) on errors 09h, 12h, and 21h.

See Also: INT 2A/AH=01h, *INT 5C*

INT 2Ah Function 05h network

GET NETWORK RESOURCE INFORMATION

Determine the available amounts of several important network resources.

Call with:
AX 0500h

Returns:
AX reserved
BX number of network names available
CX number of commands (NCBs) available
DX number of sessions available

INT 2Ah Function 06h NETBIOS

NETWORK PRINT-STREAM CONTROL

Specify behavior of redirected network printer output.

Call with:
AH 06h
AL 01h set concatenation mode (all printer output put in one job)
02h set truncation mode (default)
printer open/close starts new print job
03h flush printer output and start new print job

Returns:
CF set on error
AX error code

Note:
Subfunction 03h is equivalent to pressing Ctrl/Alt/keypad-*.

See Also: INT 21/AX=5D08h, INT 21/AX=5D09h, INT 2F/AX=1125h

INT 2Ah Function 2001h network

UNKNOWN

The purpose of this function is not known.

Call with:
AX 2001h
additional arguments (if any) unknown

Returns:
unknown

Note:
This function is intercepted by DESQview 2.x.

INT 2Ah Function 2002h network

UNKNOWN

The purpose of this function is not known.

Call with:
AX 2002h
additional arguments (if any) unknown

Returns:
unknown

Note:
This function is called by MSDOS 3.30 APPEND.

INT 2Ah Function 2003h network

UNKNOWN

The purpose of this function is not known.

Call with:

AX 2003h
additional arguments (if any) unknown

Returns:

unknown

Note:

This function is called by MSDOS 3.30 APPEND.

INT 2Ah Function 80h network

BEGIN DOS CRITICAL SECTION

Indicate that an uninterruptible region of code is being entered.

Called with:

AH 80h
AL critical section number (00h-0Fh)

01h	DOS kernel, SHARE.EXE
02h	DOS kernel
05h	DOS 4+ IFSFUNC
06h	DOS 4+ IFSFUNC
08h	ASSIGN.COM

Notes:

- This function is normally hooked to keep track of which critical sections are in effect, rather than being called by a user program. Knowledge of the critical regions in effect is necessary to properly reenter DOS using the swappable data area returned by INT 21/AX=5D06h or INT 21/AX=5D0Bh.

See Also: INT 21/AX=5D06h, INT 21/AX=5D0Bh, INT 2A/AH=81h, INT 2A/AH=82h, INT 2A/AH=87h

- The handler should ensure that none of the critical sections are reentered, usually by suspending a task which attempts to reenter an active critical section.
- Critical section 01h is apparently used to maintain the integrity of DOS, SHARE, and netork data structures.
- Critical section 02h ensures that no multitasking occurs while DOS is calling a device driver.

INT 2Ah Function 81h network

END DOS CRITICAL SECTION

Indicate that an uninterruptible region of code is being left.

Called with:

AH	81h
AL	critical section number (00h-0Fh) (see INT 2A/AH=80h)

Note:
This function is normally hooked rather than called by a user program.

See Also: INT 2A/AH=80h, INT 2A/AH=82h, INT 2A/AH=87h

INT 2Ah Function 82h network

END CRITICAL SECTIONS 0 THROUGH 7

Clean up any DOS critical section flags which may have been left set by an aborted process or DOS function call.

Called with:

AH	82h

Notes:
- The INT 21h function dispatcher calls this function for DOS function 0 and DOS functions greater than 0Ch except 59h. DOS also calls this function on process termination.

- This function is normally hooked rather than called by a user program.

See Also: INT 2A/AH=81h

INT 2Ah Function 84h network

KEYBOARD BUSY LOOP

This is a hook to let other work proceed while waiting for keyboard input.

Called with:

AH	84h

Note:
similar to DOS's INT 28h

See Also: INT 28h

INT 2Ah Function 87h network

CRITICAL SECTION

Specify the start or end of a critical section of code.

Call with:

AH	87h
AL	00h start
	01h end

Note:
This function is called by PRINT.COM.

See Also: INT 2A/AH=80h, INT 2A/AH=81h

INT 2Ah Function 89h network

UNKNOWN

The purpose of this function is not known.

Call with:

AH 89h
AL unknown (ASSIGN uses 08h)
additional arguments (if any) unknown

Returns:

unknown

INT 2Ah Function C2h network

UNKNOWN

The purpose of this function is not known.

Call with:

AH C2h
AL subfunction
07h unknown
08h unknown
BX 0001h
additional arguments (if any) unknown

Returns:

unknown

Note:

This function is called by DOS 3.30 APPEND.

INT 2Bh DOS 2+

UNUSED

This vector points at an IRET instruction under DOS 2.0 through 4.01.

INT 2Ch DOS 2+

UNUSED

This vector points at an IRET instruction under DOS 2.0 through 4.01.

INT 2Dh DOS 2+

UNUSED

This vector points at an IRET instruction under DOS 2.0 through 4.01.

INT 2Eh DOS 2+

EXECUTE COMMAND

Force COMMAND.COM to execute a command as if it were typed from the keyboard.

Call with:

DS:SI pointer to counted CR-terminated command string

Notes:

- The top-level COMMAND.COM executes the command.
- All registers including SS and SP are destroyed as in INT 21/AH=4Bh.
- Since COMMAND.COM processes the string as if typed from the keyboard, the transient portion needs to be present, and the calling program must ensure that sufficient memory to load the transient portion can be allocated by DOS if necessary.

INT 2Fh Function 00h DOS 2.x only

PRINT.COM - UNKNOWN

The purpose of this function is not known.

Call with:
AH 00h
additional arguments (if any) unknown

Returns:
unknown

Notes:
- DOS 2.x PRINT.COM does not chain to the previous INT 2Fh handler.
- Values of AH other than 00h or 01h cause PRINT to return in AH the number of files in the queue.

See Also: INT 2F/AH=01h

INT 2Fh Function 0080h DOS 3.1+

PRINT.COM - GIVE PRINT A TIME SLICE

Allow PRINT to execute for a while.

Call with:
AX 0080h

Returns:
after PRINT executes

INT 2Fh Function 01h DOS 2.x only

PRINT.COM - UNKNOWN

The purpose of this function is not known.

Call with:
AH 01h
additional arguments (if any) unknown

Returns:
unknown

Notes:
- DOS 2.x PRINT.COM does not chain to the previous INT 2Fh handler
- Values of AH other than 00h or 01h cause PRINT to return in AH the number of files in the queue.

See Also: INT 2F/AH=00h

INT 2Fh Function 0106h DOS 3.3+

PRINT.COM - CHECK IF ERROR ON OUTPUT DEVICE

Determine whether the PRINT output device is currently in an error state.

Call with:
AX 0106h

Returns:
CF set on error
    AX error code
    DS:SI pointer to device driver header
CF clear if successful
    AX 0000h

See Also: *INT 2F/AX=0104h*

INT 2Fh Function 0200h — PC LAN PROGRAM REDIR/REDIRIFS

INSTALLATION CHECK

Determine whether the PC LAN Program redirector is installed.

Call with:
AX 0200h

Returns:
AL FFh if installed

INT 2Fh Functions 0201h-0204h — PC LAN PROGRAM REDIR/REDIRIFS

UNKNOWN

The purpose of these functions is not known.

Call with:
AH 02h
AL subfunction

Returns:
unknown, probably nothing

Notes:
- These functions are called by DOS 3.3+ PRINT.COM.
- AL=01h and 02h appear to be inverse functions, as well as 03h and 04h.

INT 2Fh Function 0500h — DOS 3+

CRITICAL ERROR HANDLER - INSTALLATION CHECK

Determine whether code to expand an error number into the corresponding error message has been loaded.

Called with:
AX 0500h

Returns:

AL 00h not installed, OK to install
01h not installed, can't install
FFh installed

Note:

This set of functions allows a user program to partially or completely override the default critical error messages in COMMAND.COM.

See Also: INT 24

INT 2Fh Function 05h DOS 3+

CRITICAL ERROR HANDLER - EXPAND ERROR INTO STRING

Convert an error number into the corresponding error message.

Called with:

AH 05h

DOS 3.x

AL extended error code (not zero)

DOS 4.x

AL error type
01h DOS extended error code
02h parameter error
BX error code

Returns:

CF clear if successful
ES:DI pointer to ASCIZ error message (read-only)
AL *unknown*
CF set if error code can't be converted to string

Notes:

- This function is called at the start of COMMAND.COM's default critical error handler if INT 2F/AX=0500h indicates that a handler is installed, allowing partial or complete overriding of the default error messages.
- Subfunction 02h is called by many of the DOS 4 external commands.

See Also: INT 2F/AX=122Eh, INT 24

INT 2Fh Function 0600h DOS 3+

ASSIGN - INSTALLATION CHECK

Determine whether ASSIGN has been loaded.

Call with:
AX 0600h

Returns:
AL nonzero if installed

INT 2Fh Function 0601h DOS 3+

ASSIGN - GET MEMORY SEGMENT

Return a pointer to the drive translation table used by ASSIGN.

Call with:
AX 0601h

Returns:
ES segment of ASSIGN work area and assignment table

Note:
Under DOS 3.1+, the 26 bytes starting at ES:0103h specify which drive each of A: to Z: is mapped to. Initially set to 01h 02h 03h....

INT 2Fh Function 0800h DOS 3.2+

DRIVER.SYS SUPPORT - INSTALLATION CHECK

Determine whether the DRIVER.SYS support is present.

Call with:
AX 0 800h

Returns:

AL 00h not installed, OK to install
01h not installed, not OK to install
FFh installed

INT 2Fh Function 0801h DOS 3.2+

DRIVER.SYS SUPPORT - ADD NEW BLOCK DEVICE

Add a new logical drive alias for an existing physical drive.

Call with:

AX 0801h
DS:DI pointer to drive data table (see INT 2F/AX=0803h)

Notes:

- Scans the internal list of drive data tables, copying and modifying the drive description flags word for tables referencing same physical drive.
- The new data table is appended to the chain of tables.

See Also: INT 2F/AX=0803h

INT 2Fh Function 0802h DOS 3.2+

DRIVER.SYS SUPPORT - EXECUTE DEVICE DRIVER REQUEST

Execute the specified device driver request for a drive alias established by INT 2F/AX=0801h.

Call with:

AX 0802h
ES:BX pointer to device driver request header (see below)

Returns:

request header updated as per requested operation

Format of device driver request header:

Offset	Size	Description
00h	BYTE	length of request header
01h	BYTE	subunit within device driver
02h	BYTE	command code (see below)
03h	WORD	status (filled in by device driver)
		bit 15: error
		bits 14-10: reserved
		bit 9: busy
		bit 8: done
		bits 7-0: error code if bit 15 set (see below)
05h	8 BYTEs	reserved (unused by DOS <= 3.3)
command code 00h		
0Dh	BYTE	number of units (set by driver)
0Eh	DWORD	address of first free byte following driver (set by driver)
12h	DWORD	pointer to BPB array (set by block drivers only)
16h	BYTE	(DOS 3+) drive number for first unit of block driver (0=A)
command code 01h		
0Dh	BYTE	media descriptor
0Eh	BYTE	returned status
		00h don't know
		01h media has not changed
		FFh media has been changed
0Fh	DWORD	(DOS 3+) pointer to previous volume ID if OPEN/CLOSE/RM bit in device header set and disk changed (set by driver)
command code 02h		
0Dh	BYTE	media descriptor
0Eh	DWORD	transfer address
		pointer to scratch sector if NON-IBM FORMAT bit in device header set
		pointer to first FAT sector otherwise
12h	DWORD	pointer to BPB (set by driver)
command codes 03h,0Ch		
0Dh	BYTE	media descriptor (block devices only)
0Eh	DWORD	transfer address
12h	WORD	byte count (character devices) or sector count (block devices)
14h	WORD	starting sector number (block devices only)

command codes 04h,08h,09h

0Dh	BYTE	media descriptor (block devices only)
0Eh	DWORD	transfer address
12h	WORD	byte count (character devices) or sector count (block devices)
14h	WORD	starting sector number (block devices only)
16h	DWORD	(DOS 3+) pointer to volume ID if error 0Fh returned

command code 05h

0Dh	BYTE	byte read from device if BUSY bit clear on return

command codes 06h,07h,0Ah,0Bh

no further fields

command code 10h

0Dh	BYTE	unused
0Eh	DWORD	transfer address
12h	WORD	byte count

command code 13h

0Dh	BYTE	category code 00h unknown 01h COMn: 03h CON 05h LPTn: 08h disk
0Eh	BYTE	function code
0Fh	*DWORD*	*apparently unused in DOS 3.3*
13h	DWORD	pointer to parameter block from INT 21/AX=440Dh

Values for command code:

00h INIT
01h MEDIA CHECK (block devices)
02h BUILD BPB (block devices)
03h IOCTL INPUT
04h INPUT
05h NONDESTRUCTIVE INPUT, NO WAIT (character devices)
06h INPUT STATUS (character devices)
07h INPUT FLUSH (character devices)
08h OUTPUT
09h OUTPUT WITH VERIFY
0Ah OUTPUT STATUS (character devices)
0Bh OUTPUT FLUSH (character devices)
0Ch IOCTL OUTPUT

0Dh (DOS 3+) DEVICE OPEN
0Eh (DOS 3+) DEVICE CLOSE
0Fh (DOS 3+) REMOVABLE MEDIA (block devices)
10h (DOS 3+) OUTPUT UNTIL BUSY (character devices)
11h unused
12h unused
13h (DOS 3.2+) GENERIC IOCTL
14h unused
15h unused
16h unused
17h (DOS 3.2+) GET LOGICAL DEVICE
18h (DOS 3.2+) SET LOGICAL DEVICE
80h (CD-ROM) READ LONG
81h (CD-ROM) reserved
82h (CD-ROM) READ LONG PREFETCH
83h (CD-ROM) SEEK
84h (CD-ROM) PLAY AUDIO
85h (CD-ROM) STOP AUDIO
86h (CD-ROM) WRITE LONG
87h (CD-ROM) WRITE LONG VERIFY
88h (CD-ROM) RESUME AUDIO

Values for error code:

00h write-protect violation
01h unknown unit
02h drive not ready
03h unknown command
04h CRC error
05h bad drive request structure length
06h seek error
07h unknown media
08h sector not found
09h printer out of paper
0Ah write fault
0Bh read fault
0Ch general failure
0Dh reserved
0Eh reserved
0Fh invalid disk change

INT 2Fh Function 0803h DOS 4+

DRIVER.SYS SUPPORT - GET DRIVE DATA TABLE LIST

Return a pointer to the first in a list of drive data tables describing the layout of the logical drives supported by the combination of the default disk device driver and aliases established with DRIVER.SYS.

Call with:

AX 0803h

Returns:

DS:DI pointer to first drive data table in list

Format of DOS 3.3 drive data table:

Offset	Size	Description
00h	DWORD	pointer to next table
04h	BYTE	physical unit number (for INT 13h)
05h	BYTE	logical drive number
06h	19 BYTEs	BIOS Parameter Block (*See also* INT 21/AH=53h)

Offset	Size	Description
00h	WORD	bytes per sector
02h	BYTE	sectors per cluster, FFh if unknown
03h	WORD	number of reserved sectors
05h	BYTE	number of FATs
06h	WORD	number of root dir entries
08h	WORD	total sectors
0Ah	BYTE	media descriptor, 00h if unknown
0Bh	WORD	sectors per FAT
0Dh	WORD	sectors per track
0Fh	WORD	number of heads
11h	WORD	number of hidden sectors

19h	*BYTE*	*unknown*
1Ah	WORD	number of DEVICE OPEN calls without corresponding DEVICE CLOSE
1Ch	11 BYTEs	volume label or "NO NAME" if none
27h	*BYTE*	*terminating null for volume label*
28h	BYTE	device type (see *INT 21/AX=440Dh*)
29h	WORD	bit flags describing drive
		bit 0: fixed media

		bit 1: door lock supported
		bit 2: used in determining BPB to set for *INT 21/AX=440Dh*
		bit 3: all sectors in a track are the same size
		bit 4: physical drive has multiple logical units
		bit 5: current logical drive for physical drive
		bit 6: unknown
		bit 7: unknown
		bit 8: related to disk change detection
2Bh	WORD	number of cylinders
2Dh	19 BYTEs	BIOS Parameter Block for highest capacity supported
40h	*3 BYTEs*	*unknown*
43h	*9 BYTEs*	*file system type, default "NO NAME "*
4Ch	*BYTE*	*terminating null for filesystem type*
4Dh	DWORD	time of last access in clock ticks (FFFFFFFFh if never)
		removable media only

Format of DOS 4.01 drive data table:

Offset	Size	Description
00h	DWORD	pointer to next table
04h	BYTE	physical unit number (for INT 13h)
05h	BYTE	logical drive number
06h	19 BYTEs	BIOS Parameter Block (see also INT 21/AH=53h)

Offset	Size	Description
00h	WORD	bytes per sector
02h	BYTE	sectors per cluster, FFh if unknown
03h	WORD	number of reserved sectors
05h	BYTE	number of FATs
06h	WORD	number of root dir entries
08h	WORD	total sectors
0Ah	BYTE	media descriptor, 00h if unknown
0Bh	WORD	sectors per FAT
0Dh	WORD	sectors per track
0Fh	WORD	number of heads
11h	WORD	number of hidden sectors

19h	*9 BYTEs*	*unknown*
22h	BYTE	device type (see *INT 21/AX=440Dh*)
23h	WORD	bit flags describing drive
		bit 0: fixed media
		bit 1: door lock supported

		bit 2: unknown
		bit 3: all sectors in a track are the same size
		bit 4: physical drive has multiple logical units
		bit 5: current logical drive for physical drive
		bits 6-15: unknown
25h	WORD	number of cylinders
27h	19 BYTEs	BIOS Parameter Block for highest capacity supported
3Ah	*13 BYTEs*	*unknown*
47h	DWORD	time of last access in clock ticks (FFFFFFFFh if never)
4Bh	11 BYTEs	volume label or "NO NAME" if none
56h	*BYTE*	*terminating null for volume label*
57h	DWORD	serial number
5Bh	8 BYTEs	filesystem type ("FAT12" or "FAT16")
63h	*BYTE*	*terminating null for filesystem type*

See Also: INT 2F/AX=0801h

INT 2Fh Function 1000h DOS 3+

SHARE - INSTALLATION CHECK

Determine whether SHARE has been loaded.

Call with:

AX 1000h

Returns:

AL	00h	not installed, OK to install
	01h	not installed, not OK to install
	FFh	installed

BUG:

Values of AL other than 00h put DOS 3.x SHARE into an infinite loop
(08E9: OR AL,AL
08EB: JNZ 08EB the buggy instruction for DOS 3.3)
Values of AL other than described here put PCDOS 4.0x into the same loop (the buggy instructions are the same).

See Also: INT 21/AH=52h

INT 2Fh Function 1040h DOS 4+

SHARE - UNKNOWN

The purpose of this function is not known.

Call with:
AX 1040h
additional arguments (if any) unknown

Returns:
unknown

INT 2Fh Function 1080h DOS 4+

SHARE - CLEAR UNKNOWN FLAG

The purpose of the flag this function clears is not known.

Call with:
AX 1080h

Returns:
AL F0h function supported

INT 2Fh Function 1081h DOS 4+

SHARE - SET UNKNOWN FLAG

The purpose of the flag this function sets is not known.

Call with:
AX 1081h

Returns:
AL F0h function supported

INT 2Fh Function 1100h DOS 3.1+

NETWORK REDIRECTOR - INSTALLATION CHECK

Determine whether a network redirector using the DOS kernel network hooks is installed.

Called with:

AX 1100h

Returns:

AL 00h not installed, OK to install
01h not installed, not OK to install
FFh installed

Notes:

- This function is called by the DOS 3.1+ kernel.
- In DOS 4+, the 11xx calls are all in IFSFUNC.EXE, not in the PC LAN Program redirector.

INT 2Fh Function 1101h DOS 3.1+

NETWORK REDIRECTOR - REMOVE REMOTE DIRECTORY

Remove a directory on a network or installable file system drive.

Called with:

AX	1101h
SS	set to DOS CS
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified directory name
SDA CDS pointer	pointer to CDS for drive with dir

Returns:

CF set on error
AX DOS error code (see *INT 21/AH=59h*)

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 2F/AX=1103h, INT 2F/AX=1105h, *INT 21/AH=3Ah*, INT 21/AH=60h

INT 2Fh Function 1102h DOS 4+

IFSFUNC.EXE - REMOVE REMOTE DIRECTORY

Remove a directory on a network or installable file system drive.

Called with:

AX	1102h
SS	set to DOS CS
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified directory name
SDA CDS pointer	pointer to CDS for drive with directory

Returns:

CF set on error

AX DOS error code (see *INT 21/AH=59h*)

Note:

This function appears to be identical to INT 2F/AX=1101h.

See Also: INT 2F/AX=1101h, INT 21/AH=60h

INT 2Fh Function 1103h DOS 3.1+

NETWORK REDIRECTOR - MAKE REMOTE DIRECTORY

Create a new directory on a network or installable file system drive.

Called with:

AX	1103h
SS	set to DOS CS
SDA first filename pointer	pointer to fully-qualified directory name
SDA CDS pointer	pointer to CDS for drive with directory

Returns:

CF set on error

AX DOS error code (see *INT 21/AH=59h*)

Note:
This function is called by the DOS 3.1+ kernel.

See Also: INT 2F/AX=1101h, INT 2F/AX=1105h, *INT 21/AH=39h* , **INT 21/AH=60h**

INT 2Fh Function 1104h DOS 4+

IFSFUNC.EXE - MAKE REMOTE DIRECTORY

Create a new directory on a network or installable file system drive.

Called with:

AX	1104h
SS	set to DOS CS
SDA first filename pointer	pointer to fully-qualified directory name
SDA CDS pointer	pointer to CDS for drive with dir

Returns:
CF set on error
AX DOS error code (see *INT 21/AH=59h*)

Note:
This function appears to be identical to INT 2F/AX=1103h.

See Also: INT 2F/AX=1103h, INT 21/AH=60h

INT 2Fh Function 1105h DOS 3.1+

NETWORK REDIRECTOR - CHDIR

Change the current directory on a network or installable file system drive.

Called with:

AX	1105h
SS	set to DOS CS
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified directory name
SDA CDS pointer	pointer to CDS for drive with dir

Returns:
CF set on error
AX DOS error code (see *INT 21/AH=59h*)

Note:
This function is called by the DOS 3.1+ kernel.

See Also: INT 2F/AX=1101h, INT 2F/AX=1103h, *INT 21/AH=3Bh*, INT 21/AH=60h

INT 2Fh Function 1106h DOS 3.1+

NETWORK REDIRECTOR - CLOSE REMOTE FILE

Close a file which was opened on a network or installable file system drive.

Called with:

AX	1106h
ES:DI	pointer to SFT
SFT DPB field	DPB of drive with file

additional arguments (if any) unknown

Returns:
CF set on error
AX DOS error code (see *INT 21/AH=59h*)
CF clear if successful

Note:
This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AH=3Eh, INT 2F/AX=1201h, INT 2F/AX=1227h

INT 2Fh Function 1107h DOS 3.1+

NETWORK REDIRECTOR - COMMIT REMOTE FILE

Update the directory entry and flush disk buffers for a file on a network or installable file system drive.

Called with:

AX	1107h

ES:DI	pointer to SFT
SFT DPB field	DPB of drive with file

additional arguments (if any) unknown

Returns:

CF set on error
    AX DOS error code

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AH=68h, INT 21/AX=5D01h

INT 2Fh Function 1108h DOS 3.1+

NETWORK REDIRECTOR - READ FROM REMOTE FILE

Read data from a file opened on a network or installable file system drive.

Called with:

AX	1108h
ES:DI	pointer to SFT
SFT DPB field	DPB of drive with file
CX	number of bytes
SS	set to DOS CS
SDA DTA field	pointer to buffer containing data

Returns:

CF set on error
CF clear if successful
    CX number of bytes read

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 2F/AX=1109h, INT 2F/AX=1229h, INT 21/AH=3Fh, INT 21/AX=5D06h

INT 2Fh Function 1109h DOS 3.1+

NETWORK REDIRECTOR - WRITE TO REMOTE FILE

Write data to a file opened on a network or installable file system drive.

Called with:

AX	1109h
ES:DI	pointer to SFT
SFT DPB field	DPB of drive with file
CX	number of bytes
SS	set to DOS CS
SDA DTA field	pointer to buffer for data

Returns:

CF set on error
CF clear if successful
    CX number of bytes written

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 2F/AX=1107h, INT 2F/AX=1108h, *INT 21/AH=40h*, INT 21/AX=5D06h

INT 2Fh Function 110Ah DOS 3.1+

NETWORK REDIRECTOR - LOCK REGION OF FILE

Request that no other processes be allowed access to a portion of the specified file.

Called with:

AX	110Ah
BX	file handle
CX:DX	starting offset
SI	high word of size
STACK	WORD low word of size
ES:DI	pointer to SFT
SFT DPB field	DPB of drive with file
SS	set to DOS CS

Returns:

CF set on error

AL	DOS error code (see *INT 21/AH=59h*)
STACK	unchanged

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 2F/AX=110Bh, INT 21/AH=5Ch

INT 2Fh Function 110Bh DOS 3.1+

NETWORK REDIRECTOR - UNLOCK REGION OF FILE

Allow other processes to access the specified portion of the file.

Called with:

AX	110Bh
BX	file handle
CX:DX	starting offset
SI	high word of size
STACK	WORD low word of size
ES:DI	pointer to SFT for file
SFT DPB field	DPB of drive with file

Returns:

CF set on error

AL	DOS error code (see *INT 21/AH=59h*)
STACK	unchanged

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 2F/AX=110Ah, *INT 21/AH=5Ch*

INT 2Fh Function 110Ch DOS 3.1+

NETWORK REDIRECTOR - GET DISK SPACE

Get information on allocation size and disk space (free and total) for a network or installable file system drive.

Called with:

AX	110Ch
ES:DI	pointer to CDS for desired drive

Returns:

AL	sectors per cluster
AH	*unknown*
BX	total clusters
CX	bytes per sector
DX	number of available clusters

Note:
This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AH=36h

INT 2Fh Function 110Dh DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known, but appears to be related to file attributes.

Called with:

AX	110Dh
SDA first filename pointer	pointer to filename

additional arguments (if any) unknown

Returns:
unknown

Note:
This function appears to be similar to INT 2F/AX=110Fh (Get Attributes).

See Also: INT 2F/AX=110Fh

INT 2Fh Function 110Eh DOS 3.1+

NETWORK REDIRECTOR - SET REMOTE FILE'S ATTRIBUTES

Change the attributes of a file on a network or installable file system drive.

Called with:

AX	110Eh
SS	set to DOS CS
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified file name
SDA CDS pointer	CDS for drive with file
STACK	WORD new file attributes

Returns:

CF set on error

AX DOS error code (see *INT 21/AH=59h*)

CF clear if successful

STACK unchanged

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 2F/AX=110Fh, INT 21/AX=4301h, INT 21/AH=60h

INT 2Fh Function 110Fh DOS 3.1+

NETWORK REDIRECTOR - GET REMOTE FILE'S ATTRIBUTES

Get the attributes of a file on a network or installable file system drive.

Called with:

AX	110Fh
SS	set to DOS CS
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified file name
SDA CDS pointer	CDS for drive with file

Returns:

CF set on error
    AX DOS error code (see *INT 21/AH=59h*)
CF clear if successful
    AX file attributes

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 2F/AX=110Eh, INT 21/AX=4300h, INT 21/AH=60h

INT 2Fh Function 1110h DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known, but appears to be related to file attributes.

Called with:

AX	1110h
SDA first filename pointer	pointer to fully qualifiled filename

additional arguments (if any) unknown

Returns:

unknown

Note:

This function appears to be similar to INT 2F/AX=110Eh (Set Attributes).

See Also: INT 2F/AX=110Eh, INT 21/AH=60h

INT 2Fh Function 1111h DOS 3.1+

NETWORK REDIRECTOR - RENAME REMOTE FILE

Change the name of a file on a network or installable file system drive.

Called with:

AX	1111h
SS	set to DOS CS

DS	set to DOS CS
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified current file name
SDA second filename pointer	offset of SDA new file name
SDA second filename buffer	fully qualified new file name
SDA CDS pointer	CDS for drive with file

Returns:

CF set on error

AX DOS error code (see *INT 21/AH=59h*)

CF clear if successful

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AH=56h, INT 21/AH=60h

INT 2Fh Function 1112h DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known.

Called with:

AX	1112h
SS	set to DOS CS
DS	set to DOS CS
SDA first filename pointer	pointer to fully qualified filename

additional arguments (if any) unknown

Returns:

unknown

See Also: INT 2F/AX=1111h, INT 21/AH=60h

INT 2Fh Function 1113h DOS 3.1+

NETWORK REDIRECTOR - DELETE REMOTE FILE

Remove a file from a network or installable file system drive.

Called with:

AX	1113h
SS	set to DOS CS
DS	set to DOS CS
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified file name
SDA CDS pointer	CDS for drive with file

Returns:

CF set on error
    AX DOS error code (see *INT 21/AH=59h*)
CF clear if successful

Note:

This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AH=41h, INT 21/AH=60h

INT 2Fh Function 1114h DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known.

Called with:

AX	1114h
SDA first filename pointer	pointer to fully-qualified filename

additional arguments (if any) unknown

Returns:

unknown

See Also: INT 2F/AX=1113h, INT 21/AH=60h

INT 2Fh Function 1115h DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known.

Called with:

AX	1115h
SS	set to DOS CS
ES:DI	*pointer to SFT*

additional arguments (if any) unknown

Returns:

unknown

See Also: INT 2F/AX=112Eh

INT 2Fh Function 1116h DOS 3.1+

NETWORK REDIRECTOR - OPEN EXISTING REMOTE FILE

Prepare for access to an existing file located on a network drive.

Called with:

AX	1116h
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified file name
SDA CDS pointer	CDS for drive with file
ES:DI	pointer to uninitialized SFT
SS	set to DOS CS
STACK	*WORD file open mode*

additional arguments (if any) unknown

Returns:

CF set on error
    AX DOS error code (see *INT 21/AH=59h)*
CF clear if successful
    SFT completed
STACK unchanged

Note:
This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AH=3Dh, INT 21/AH=60h, INT 2F/AX=1106h, INT 2F/AX=1117h, INT 2F/AX=1118h

INT 2Fh Function 1117h DOS 3.1+

NETWORK REDIRECTOR - CREATE/TRUNCATE REMOTE FILE

Create a file on a network drive, or truncate an existing file to zero length.

Called with:

AX	1117h
ES:DI	pointer to uninitialized SFT
SS	set to DOS CS
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified file name
SDA CDS pointer	CDS for drive with file
STACK	*WORD file creation mode*

Returns:

CF set on error
  AX DOS error code (see *INT 21/AH=59h*)
CF Clear if succesful
  SFT completed
STACK unchanged

Notes:

- This function is called by the DOS 3.1+ kernel.
- INT 2F/AX=1117h appears to be identical in operation, except that it is called for drives which do not have a current directory structure (SDA CDS pointer has offset FFFFh).

See Also: *INT 21/AH=3Ch*, INT 21/AH=60h, INT 2F/AX=1106h, INT 2F/AX=1116h, INT 2F/AX=1118h

INT 2Fh Function 1118h DOS 3.1+

NETWORK REDIRECTOR - CREATE/TRUNCATE FILE

Create a file on a drive which does not have a current directory sturcture.

Called with:

AX	1118h
ES:DI	pointer to uninitialized SFT
SS	set to DOS CS
SDA first filename pointer	pointer to fully-qualified filename
STACK	*WORD file creation mode*

Returns:

unknown

STACK unchanged

Notes:

- This function is called by the DOS 3.1+ kernel when creating a file on a drive whose CDS pointer has offset FFFFh.
- INT 2F/AX=1117h is apparently equivalent to this function for remote drives which have a current directory structure.

See Also: INT 21/AH=60h, INT 2F/AX=1116h, INT 2F/AX=1117h

INT 2Fh Function 1119h DOS 3.1+

NETWORK REDIRECTOR - UNKNOWN

The purpose of this function is not known.

Called with:

AX 1119h

additional arguments (if any) unknown

Returns:

unknown

Notes:

- This function is called by the DOS 3.1+ kernel.
- DOS 4.0 IFSFUNC returns with CF set and AX=0003h.

INT 2Fh Function 111Ah DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known.

Called with:

AX 111Ah
additional arguments (if any) unknown

Returns:

CF set
AX error code (03h for DOS 4.01 IFSFUNC)

INT 2Fh Function 111Bh DOS 3.1+

NETWORK REDIRECTOR - FINDFIRST

Begin a directory search on a network or installable file system drive.

Called with:

AX	111Bh
SS	set to DOS CS
DS	set to DOS CS
SDA search data block	uninitialized 21-byte findfirst search data (see INT 21/AH=4Eh)
SDA first filename pointer	offset of SDA first filename buffer
SDA first filename buffer	fully qualified file name
SDA CDS pointer	CDS for drive with file

Returns:

CF set on error
    AX DOS error code (see *INT 21/AH=59h*)
CF clear if successful

SDA search data block completed (bit 7 of first byte must be set)
SDA file found field: standard directory entry for file

Note:
This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AH=4Eh, INT 21/AH=60h,INT 2F/AX=111Ch

INT 2Fh Function 111Ch DOS 3.1+

NETWORK REDIRECTOR - FINDNEXT

Continue a directory search on a network or installable file system drive.

Called with:

AX	111Ch
SS	set to DOS CS
DS	set to DOS CS
SDA search data block	findfirst search data from initial INT 2F/AX=111Bh

Returns:
CF set on error
AX DOS error code (see *INT 21/AH=59h*)
CF clear if successful
SDA search data block completed (bit 7 of first byte must be set)
SDA found file field: standard directory entry for file

Note:
This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AH=4Fh, INT 2F/AX=111Bh

INT 2Fh Function 111Dh DOS 3.1+

NETWORK REDIRECTOR - CLOSE ALL REMOTE FILES FOR PROCESS

Called with:

AX	111Dh
SS	set to DOS CS

additional arguments (if any) unknown

Returns:

unknown

Note:

This function is called by the DOS 3.1+ kernel, and closes all FCBs opened by the process (among other actions).

INT 2Fh Function 111Eh DOS 3.1+

NETWORK REDIRECTOR - DO REDIRECTION

Various subfunctions allow control of network redirection.

Called with:

AX 111Eh
SS set to DOS CS
STACK WORD function to execute
5F00h get redirection mode
BL type (03h printer, 04h disk)
Returns:
BH state (00h off, 01h on)
5F01h set redirection mode
BL type (03h printer, 04h disk)
BH state (00h off, 01h on)
5F02h get redirection list entry
BX redirection list index
DS:SI pointer to 16-byte local device name buffer
ES:DI pointer to 128-byte network name buffer
5F03h redirect device
BL device type (see *INT 21/AX=5F03h*)
CX stored parameter value
DS:SI pointer to ASCIZ source device name
ES:DI pointer to destination ASCIZ network path + ASCIZ password
5F04h cancel redirection
DS:SI pointer to ASCIZ device name or network path
5F05h get redirection list extended entry
BX redirection list index
DS:SI pointer to buffer for ASCIZ source device name
ES:DI pointer to buffer for destination ASCIZ network path

Returns:
BH status flag
BL type (03h printer, 04h disk)
CX stored parameter value
BP NETBIOS local session number
5F06h appears to be similar to 5F05h

Returns:
CF set on error
AX error code (see *INT 21/AH=59h*)
STACK unchanged

Note:
This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AX=5F00h, INT 21/AX=5F01h, INT 21/AX=5F02h, INT 21/AX=5F03h, INT 21/AX=5F04h, INT 21/AX=5F05h, INT 21/AX=5F06h

INT 2Fh Function 111Fh DOS 3.1+

NETWORK REDIRECTOR - PRINTER SETUP

Subfunctions allow getting or setting the network printer setup string or mode.

Called with:
AX 111Fh
STACK WORD function
5E02h set printer setup
5E03h get printer setup
5E04h set printer mode
5E05h get printer mode

Returns:
CF set on error
AX error code (see *INT 21/AH=59h*)
STACK unchanged

Note:
This function is called by the DOS 3.1+ kernel.

See Also: *INT 21/AX=5E02h, INT 21/AX=5E03h,* INT 21/AX=5E04h, INT 21/AX=5E05h

INT 2Fh Function 1120h DOS 3.1+

NETWORK REDIRECTOR - FLUSH ALL DISK BUFFERS

Force an immediate update of the network or installable file system drives from disk buffers which have not yet been written out.

Called with:

AX	1120h
DS	set to DOS CS

additional arguments (if any) unknown

Returns:

CF clear (successful)

Notes:

- This function is called by the DOS 3.1+ kernel.
- The current directory structure array pointer and LASTDRIVE= entries of the DOS list of lists are used by the DOS 4 IFSFUNC handler for this call.

See Also: *INT 21/AH=0Dh*, INT 21/AX=5D01h

INT 2Fh Function 1121h DOS 3.1+

NETWORK REDIRECTOR - SEEK FROM END OF REMOTE FILE

Called with:

AX	1121h
CX:DX	offset (in bytes) from end
ES:DI	pointer to SFT
SFT DPB field	pointer to DPB of drive containing file
SS	set to DOS CS

Returns:

CF set on error

AL	DOS error code (see *INT 21/AH=59h*)

CF clear if successful

DX:AX	new file position

Note:
This function is called by the DOS 3.1+ kernel.

See Also: *INT 21/AH=42h,* INT 2F/AX=1228h

INT 2Fh Function 1122h DOS 3.1+

NETWORK REDIRECTOR - PROCESS TERMINATION HOOK

Inform the network that a process has terminated.

Called with:

AX	1122h
SS	set to DOS CS

additional arguments (if any) unknown

Returns:
unknown

Note:
This function is called by the DOS 3.1+ kernel.

INT 2Fh Function 1123h DOS 3.1+

NETWORK REDIRECTOR - QUALIFY REMOTE FILENAME

Convert a name into a absolute pathname with any network redirections resolved.

Called with:

AX	1123h
DS:SI	pointer to ASCIZ filename to canonicalize
ES:DI	pointer to 128-byte buffer for qualified name

Returns:
CF set if not resolved

Note:
The DOS 3.1+ kernel calls this function first when it attempts to resolve a filename (unless inside an INT 21/AX=5D00h server call); if this fails, DOS resolves the name locally.

See Also: INT 21/AH=60h, INT 2F/AX=1221h

INT 2Fh Function 1124h DOS 3.1+

NETWORK REDIRECTOR - UNKNOWN

The purpose of this function is not known.

Called with:

AX 1124h
ES:DI pointer to SFT
SS set to DOS CS
additional arguments (if any) unknown

Returns:

CX unknown

Note:

This function is called by the DOS 3.1+ kernel.

INT 2Fh Function 1125h DOS 3.1+

NETWORK REDIRECTOR - REDIRECTED PRINTER MODE

Set or determine the state of print streams for the network printer.

Called with:

AX 1125h
STACK WORD subfunction
5D07h get print stream state
Returns:
DL current state
5D08h set print stream state
DL new state
5D09h finish print job

Returns:

CF set on error
AX error code (see *INT 21/AH=59h*)
STACK unchanged

Note:
This function is called by the DOS 3.1+ kernel.

See Also: INT 21/AX=5D07h, INT 21/AX=5D08h, INT 21/AX=5D09h

INT 2Fh Function 1126h DOS 3.1+

NETWORK REDIRECTOR - UNKNOWN

The purpose of this function is not known.

Called with:
AX 1126h
additional arguments (if any) unknown

Returns:
CF set on error

Note:
This function is called by the DOS 3.1+ kernel.

INT 2Fh Function 1127h DOS 4+

IFSFUNC.EXE - UNUSED

This function performs no actions and returns immediately.

Called with:
AX 1127h

Returns:
CF set
AX 0001h (invalid function) (see *INT 21/AH=59h*)

INT 2Fh Function 1128h DOS 4+

IFSFUNC.EXE - UNUSED

This function performs no actions and returns immediately.

Called with:

AX 1128h

Returns:

CF set
AX 0001h (invalid function) (see *INT 21/AH=59h*)

INT 2Fh Function 1129h DOS 4+

IFSFUNC.EXE - UNUSED

This function performs no actions and returns immediately.

Called with:

AX 1129h

Returns:

CF set
AX 0001h (invalid function) (see *INT 21/AH=59h*)

INT 2Fh Function 112Ah DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known.

Called with:

AX 112Ah
DS set to DOS CS
SDA PSP field current process ID
additional arguments (if any) unknown

Returns:
unknown

Note:
This function performs an unknown action on each IFS driver which has been installed.

INT 2Fh Function 112Bh DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known.

Called with:

AX	112Bh
SS	set to DOS CS
CX	*unknown*
DX	*unknown*
STACK	WORD low byte contains function
	0Dh unknown

additional arguments (if any) unknown

Returns:
CF set on error

AX DOS error code (see *INT 21/AH=59h*)

Note:
This function is called by the DOS 4.0 kernel.

INT 2Fh Function 112Ch DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known.

Called with:

AX	112Ch
SS	set to DOS CS
SDA current SFT pointer	pointer to SFT for file

additional arguments (if any) unknown

Returns:
CF set on error
    AX DOS error code (see *INT 21/AH=59h*)
CF clear if successful

INT 2Fh Function 112Dh DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known.

Called with:
AX 112Dh
BL subfunction
    04h unknown
    Returns:
        CF clear
    else unknown
    Returns:
        CX unknown (00h or 02h for DOS 4.01)
SS set to DOS CS

Returns:
DS set to DOS CS

Note:
This function is called by the DOS 4.0 kernel.

INT 2Fh Function 112Eh DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known.

Called with:
AX 112Eh
SS set to DOS CS
DS set to DOS CS

STACK WORD unknown low byte unknown
additional arguments (if any) unknown

Returns:
CF set on error
AX DOS error code (see *INT 21/AH=59h*)
CF clear if successful
CX unknown

Note:
This function is called by the DOS 4.0 kernel.

See Also: INT 2F/AX=1115h

INT 2Fh Function 112Fh DOS 4+

IFSFUNC.EXE - UNKNOWN

The purpose of this function is not known

Called with:
AX 112Fh
SS set to DOS CS
STACK WORD function in low byte
00h unknown
DS:SI pointer to Current Directory Structure
CL drive (1=A:)
01h unknown
DS:SI unknown pointer
CL file handle
02h unknown
DS:SI pointer to Current Directory Structure
DI unknown
CX drive (1=A:)
additional arguments (if any) unknown

Returns:
CF set on error
AX DOS error code (see *INT 21/AH=59h*)

Note:
This function is called by the DOS 4.0 kernel.

See Also: INT 21/AH=6Bh

INT 2Fh Function 1130h DOS 4+

IFSFUNC.EXE - GET IFSFUNC SEGMENT

Return the segment of the resident IFSFUNC code.

Called with:
AX 1130h

Returns:
ES CS of resident IFSFUNC

INT 2Fh Function 1200h DOS 3+

INSTALLATION CHECK

Determine whether the DOS internal services are present.

Call with:
AX 1200h

Returns:
AL FFh (for compatibility with other INT 2Fh functions)

INT 2Fh Function 1201h DOS 3+

CLOSE CURRENT FILE

Close the file currently being operated on.

Call with:

AX	1201h
SS	set to DOS CS
SDA current SFT pointer	pointer to SFT for file to be closed

Returns:

BX	*unknown*
CX	*unknown*
ES:DI	pointer to SFT for file

See Also: INT 21/AH=3Eh, INT 2F/AX=1227h

INT 2Fh Function 1202h DOS 3+

GET INTERRUPT ADDRESS

Return a pointer to the interrupt vector corresponding to the given interrupt number.

Call with:

AX	1202h
STACK	WORD vector number

Returns:

ES:BX	pointer to interrupt vector
STACK	unchanged

INT 2Fh Function 1203h DOS 3+

GET DOS DATA SEGMENT

Return the segment of IBMDOS.

Call with:

AX	1203h

Returns:

DS	segment of IBMDOS

INT 2Fh Function 1204h DOS 3+

NORMALIZE PATH SEPARATOR

Convert forward slashes into backslashes.

Call with:

AX	1204h
STACK	WORD character to normalize

Returns:

AL	normalized character (forward slash turned to backslash, all other characters unchanged)
ZF	set if path separator (forward or backslash)
STACK	unchanged

INT 2Fh Function 1205h DOS 3+

OUTPUT CHARACTER

Send a single character to the standard output of the current process.

Call with:

AX	1205h
STACK	WORD character to output

Returns:

STACK	unchanged

Note:

This function can be called only from within a DOS function call.

INT 2Fh Function 1206h DOS 3+

INVOKE CRITICAL ERROR

Cause an INT 24h, performing all necessary housekeeping and return code translations.

Call with:

AX	1206h

DI	error code
BP:SI	pointer to device driver header
SS	set to DOS CS
STACK	WORD value to be passed to INT 24h in AX

Returns:

AL	0-3 for Abort, Retry, Ignore, Fail
STACK	unchanged

See Also: *INT 24*

INT 2Fh Function 1207h DOS 3+

MAKE DISK BUFFER MOST-RECENTLY USED

Move specified disk buffer to the end of the disk buffer list (which is kept in reverse order of recency of use).

Call with:

AX	1207h
DS:DI	pointer to disk buffer

Returns:
buffer moved to end of buffer list

Note:
This function can only be called from within a DOS function call.

See Also: INT 2F/AX=120Fh

INT 2Fh Function 1208h DOS 3+

DECREMENT SFT REFERENCE COUNT

Reduce the number of references to the given System File Table by one, setting the count to -1 if there are now no more references to the SFT. Used by network redirectors such as MSCDEX.

Call with:

AX	1208h
ES:DI	pointer to SFT

Returns:

AX	old value of reference count (before decrement)

Note:

The reference count is set to FFFFh to indicate no references, since 0 indicates that the SFT is not in use. In this case, the caller should set the count to zero after cleaning up.

INT 2Fh Function 1209h DOS 3+

FLUSH AND FREE DISK BUFFER

Force the given disk buffer's contents to disk if it is dirty, and then mark the buffer unused.

Call with:

AX	1209h
DS:DI	pointer to disk buffer

Note:

This function can only be called from within a DOS function call.

See Also: INT 2F/AX=120Eh, INT 2F/AX=1215h

INT 2Fh Function 120Ah DOS 3+

PERFORM CRITICAL ERROR INTERRUPT

Invoke an INT 24h, passing the appropriate values for the current drive and operation (stored in the SDA). Used by network redirectors such as MSCDEX.

Call with:

AX	120Ah
DS	set to DOS CS
SS	set to DOS CS
STACK	*WORD extended error code*

Returns:

AL	user response (00h = ignore, 01h = retry, 02h = abort, 03h = fail)
CF	clear if retry, set otherwise
STACK	unchanged

Note:
This function can only be called from within a DOS function call, as it uses various fields in the SDA.

INT 2Fh Function 120Bh DOS 3+

SIGNAL SHARING VIOLATION TO USER

Produce a critical error interrupt if an attempt was made to open file previously opened in compatibility mode with inheritance allowed.

Call with:

AX	120Bh
ES:DI	pointer to system file table entry
DS	set to DOS CS
SS	set to DOS CS
STACK	*WORD extended error code (should be 20h—sharing violation)*

Returns:

STACK unchanged
CF set if no INT 24h generated or user did not say to retry operation
    AX error code (20h) (see *INT 21/AH=59h*)
CF clear if user said to retry operation

Note:

This function can only be called from within a DOS function call.

INT 2Fh Function 120Ch DOS 3+

SET FCB FILE'S OWNER

Apparently sets the owner of the last-accessed FCB file to the calling process's ID. Used by network redirectors such as MSCDEX.

Call with:

AX	120Ch
DS	set to DOS CS
ES:DI	pointer to SFT for file

Returns:
None; AX destroyed

INT 2Fh Function 120Dh DOS 3+

GET DATE AND TIME

Return the current date and time in directory format.

Call with:

AX	120Dh
SS	set to DOS CS

Returns:

AX	current date in packed format (see *INT 21/AX=5700h*)
DX	current time in packed format (see *INT 21/AX=5700h*)

See Also: *INT 21/AH=2Ah, INT 21/AH=2Ch*

INT 2Fh Function 120Eh DOS 3+

MARK ALL DISK BUFFERS UNREFERENCED

Clear the "referenced" flag on all disk buffers. This flag is automatically set when a buffer is read or written, and is used in the buffer replacement algorithm. Unreferenced buffers are generally replaced before referenced buffers.

Call with:

AX	120Eh
SS	set to DOS CS

Returns:

DS:DI	pointer to first disk buffer

See Also: INT 2F/AX=1209h, *INT 21/AH=0Dh*

INT 2Fh Function 120Fh DOS 3+

MAKE BUFFER MOST RECENTLY USED

Move the specified disk buffer to the end of the buffer chain without flushing it to disk if it is dirty.

Call with:

AX	120Fh
DS:DI	pointer to disk buffer
SS	set to DOS CS

Returns:

DS:DI	pointer to next buffer in buffer list

See Also: INT 2F/AX=1207h

INT 2Fh Function 1210h DOS 3+

FIND UNREFERENCED DISK BUFFER

Return a pointer to the least-recently used disk buffer (if any) which has not been referenced since its first use or the last "Mark all Disk Buffers Unreferenced" call.

Call with:

AX	1210h
DS:DI	pointer to first disk buffer to check

Returns:

ZF clear if found

DS:DI	pointer to first unreferenced disk buffer

ZF set if not found (all buffers have been referenced)

See Also: INT 2F/AX=120Eh

INT 2Fh Function 1211h DOS 3+

NORMALIZE ASCIZ FILENAME

Copy the given filename, converting it to uppercase and changing forward slashes into backslashes.

Call with:

AX	1211h
DS:SI	pointer to ASCIZ filename to normalize
ES:DI	pointer to buffer for normalized filename

Returns:

ES:DI buffer filled

See Also: INT 2F/AX=121Eh, INT 2F/AX=1221h

INT 2Fh Function 1212h DOS 3+

GET LENGTH OF ASCIZ STRING

Return the length of a null-terminated character string.

Call with:

AX	1212h
ES:DI	pointer to ASCIZ string

Returns:

CX length of string

See Also: INT 2F/AX=1225h

INT 2Fh Function 1213h DOS 3+

UPPERCASE CHARACTER

Return the uppercase equivalent, using the current country's capitalization rules, of the given character.

Call with:

AX	1213h
STACK	WORD character to convert to uppercase

Returns:

AL	uppercased character
STACK	unchanged

INT 2Fh Function 1214h DOS 3+

COMPARE FAR POINTERS

Determine whether two FAR pointers are identical.

Call with:

AX	1214h
DS:SI	first pointer
ES:DI	second pointer

Returns:

ZF set if pointers are equal, ZF clear if not equal

INT 2Fh Function 1215h DOS 3+

FLUSH BUFFER

Force the contents of the specified disk buffer to be written to disk if it is dirty.

Call with:

AX	1215h
DS:DI	pointer to disk buffer

SS set to DOS CS
STACK WORD drives for which to skip buffer
ignore buffer if drive same as high byte, or bytes differ and the buffer is for a drive *other* than that given in low byte

Returns:
STACK unchanged

See Also: INT 2F/AX=1209h

INT 2Fh Function 1216h DOS 3+

GET ADDRESS OF SYSTEM FILE TABLE

Return the address of a system file table entry given its number.

Call with:
AX 1216h
BX system file table entry number

Returns:
CF clear if successful
ES:DI pointer to system file table entry
CF set if BX greater than FILES=

See Also: INT 2F/AX=1220h

INT 2Fh Function 1217h DOS 3+

SET WORKING DRIVE

Call with:
AX 1217h
SS set to DOS CS
STACK WORD drive (0=A:, 1=B:, etc)

Returns:
CF set on error (drive > LASTDRIVE)
CF clear if successful

DS:SI pointer to current directory structure for specified drive
STACK unchanged

See Also: INT 2F/AX=1219h

INT 2Fh Function 1218h DOS 3+

GET CALLER'S REGISTERS

Return a pointer to the stack frame containing the INT 21h caller's registers.

Call with:
AX 1218h

Returns:
DS:SI pointer to saved caller's AX,BX,CX,DX,SI,DI,BP,DS,ES (on stack)

Note:

The result of this function is only valid while within a DOS function call.

INT 2Fh Function 1219h DOS 3+

SET DRIVE

Call with:
AX 1219h
SS set to DOS CS
STACK WORD drive (0=default, 1=A:, etc)

Returns:
unknown
STACK unchanged

Note:
This call eventually performs the equivalent of INT 2F/AX=1217h; in addition, it builds a current directory structure if inside a server call (INT 21/AX=5D00h).

See Also: INT 2F/AX=1217h, INT 2F/AX=121Fh

INT 2Fh Function 121Ah DOS 3+

GET FILE'S DRIVE

Determine which drive a filename specifies.

Call with:

AX	121Ah
DS:SI	pointer to filename

Returns:

AL drive (0=default, 1=A:, etc, FFh=invalid)

See Also: *INT 21/AH=19h*, INT 21/AH=60h

INT 2Fh Function 121Bh DOS 3+

SET YEAR/LENGTH OF FEBRUARY

Specify the current year, and return the length of February in days after storing that length internally.

Call with:

AX	121Bh
CL	year - 1980
DS	set to DOS CS

Returns:

AL number of days in February

See Also: *INT 21/AH=2Bh*

INT 2Fh Function 121Ch DOS 3+

CHECKSUM MEMORY

Compute a checksum of the given range of memory.

Call with:

AX	121Ch

DS:SI pointer to start of memory to checksum
CX number of bytes
DX initial checksum

Returns:
DX checksum
DS:SI points beyond checksummed range
AX, CX destroyed

See Also: INT 2F/AX=121Dh

INT 2Fh Function 121Dh DOS 3+

SUM MEMORY

Add up the values of a range of bytes until the specified limit is exceeded, and return the value which caused the limit to be exceeded.

Call with:
AX 121Dh
DS:SI pointer to memory to add up
CX 0000h
DX limit

Returns:
AL byte which exceeded limit
CX number of bytes before limit exceeded
DX remainder after adding first CX bytes
DS:SI points at byte beyond the one which exceeded the limit

See Also: INT 2F/AX=121Ch

INT 2Fh Function 121Eh DOS 3+

COMPARE FILENAMES

Determine whether two filenames are identical except for case and forward/backslash differences.

Call with:
AX 121Eh

DS:SI pointer to first ASCIZ filename
ES:DI pointer to second ASCIZ filename

Returns:
ZF set if filenames equivalent, ZF clear if not

See Also: INT 2F/AX=1211h, INT 2F/AX=1221h

INT 2Fh Function 121Fh DOS 3+

BUILD CURRENT DIRECTORY STRUCTURE

Create a new Current Directory Structure for the specified drive, and return the address of the temporary storage in which it was built.

Call with:
AX 121Fh
SS set to DOS CS
STACK WORD drive letter

Returns:
ES:DI pointer to current directory structure (will be overwritten by next call)
STACK unchanged

INT 2Fh Function 1220h DOS 3+

GET JOB FILE TABLE ENTRY

Given a file handle, return the address of the entry in the Job File Table for that handle in the current process.

Call with:
AX 1220h
BX file handle

Returns:
CF set on error
    AL 6 (invalid file handle)
CF clear if successful
    ES:DI pointer to JFT entry

Note:
The byte pointed at by ES:DI contains the number of the SFT entry for the file handle, or FFh if the handle is not open.

See Also: INT 2F/AX=1216h, INT 2F/AX=1229h

INT 2Fh Function 1221h DOS 3+

CANONICALIZE FILE NAME

Given a file specification, return an absolute pathname which takes into account any renaming due to JOIN, SUBST, ASSIGN, or network redirections.

Call with:

AX	1221h
DS:SI	pointer to file name to be fully qualified
ES:DI	pointer to 128-byte buffer for resulting canonical file name
SS	set to DOS CS

Returns:
see INT 21/AH=60h

Note:
This function can only be called from within a DOS function call, and is identical to INT 21/AH=60h.

See Also: INT 21/AH=60h, INT 2F/AX=1123h

INT 2Fh Function 1222h DOS 3+

SET EXTENDED ERROR INFO

Given a set of translation records, set the error class, locus, and suggested action corresponding to the current extended error code.

Call with:

AX	1222h

SDA error code field set

SS	set to DOS CS	
SS:SI	pointer to 4-byte records	
	BYTE	error code, FFh if last record
	BYTE	error class, FFh means don't change
	BYTE	suggested action, FFh means don't change
	BYTE	error locus, FFh means don't change

Returns:

SI destroyed
SDA error class, error locus, and suggested action fields set

See Also: INT 2F/AX=122Dh, *INT 21/AH=59h*

INT 2Fh Function 1223h DOS 3+

CHECK IF CHARACTER DEVICE

Determine whether the given name is the name of a character device.

Call with:

AX 1223h
DS set to DOS CS
SS set to DOS CS

DOS 3.10-3.30

SDA+218h eight-character blank-padded name

DOS 4.0x

SDA+22Bh eight-character blank-padded name

Returns:

CF set if no character device by that name found
CF clear if found
BH low byte of device attribute word

See Also: INT 21/AX=5D06h, INT 21/AX=5D0Bh

INT 2Fh Function 1224h DOS 3+

DELAY
Perform a sharing retry delay loop.

Call with:

AX	1224h
SS	set to DOS CS

Returns:
after delay set by INT 21/AX=440Bh, unless in server call (INT 21/AX=5D00h)

Note:
The delay is dependent on the processor speed, and is skipped entirely if inside a server call (INT 21/AX=5D00h).

See Also: *INT 21/AX=440Bh*, INT 21/AH=52h

INT 2Fh Function 1225h DOS 3+

GET LENGTH OF ASCIZ STRING

Return the length of a null-terminated character string.

Call with:

AX	1225h
DS:SI	pointer to ASCIZ string

Returns:

CX	length of string

See Also: INT 2F/AX=1212h

INT 2Fh Function 1226h DOS 3.3+

OPEN FILE

Open an existing file with the specified access mode and return a file handle if successful.

Call with:

AX	1226h
CL	access mode
DS:DX	pointer to ASCIZ filename
SS	set to DOS CS

Returns:

CF set on error
    AL error code (see *INT 21/AH=59h*)
CF clear if successful
    AX file handle

Notes:

This function can only be called from within a DOS function call, and is equivalent to INT 21/AH=3Dh.

See Also: INT 2F/AX=1227h, *INT 21/AH=3Dh*

INT 2Fh Function 1227h DOS 3.3+

CLOSE FILE

Close a previously-opened file given its handle.

Call with:

AX	1227h
BX	file handle
SS	set to DOS CS

Returns:

CF set on error
    AL 06h invalid file handle

Note:
This function is equivalent to INT 21/AH=3Eh, but may only be called when already inside a DOS function call.

See Also: INT 2F/AX=1226h, *INT 21/AH=3Eh*

INT 2Fh Function 1228h — DOS 3.3+

MOVE FILE POINTER

Set the current position in the given file.

Call with:

AX	1228h
BP	4200h, 4201h, 4202h (see *INT 21/AH=42h*)
BX	file handle
CX:DX	offset in bytes
SS	set to DOS CS

Returns:

CF	set on error
AX	error code (01h,06h) (see *INT 21/AH=59h*)
CF	clear if successful
DX:AX	new absolute offset from beginning of file

Note:
This function is equivalent to INT 21/AH=42h, but may only be called when already inside a DOS function call.

See Also: *INT 21/AH=42h*

INT 2Fh Function 1229h — DOS 3.3+

READ FROM FILE

Read data from a previously-opened file.

Call with:

AX	1229h

BX	file handle
CX	number of bytes to read
DS:DX	pointer to buffer
SS	set to DOS CS

Returns:

as for INT 21/AH=3Fh

Note:

This function is equivalent to INT 21/AH=3Fh, but may only be called when already inside a DOS function call.

See Also: INT 2F/AX=1226h, *INT 21/AH=3Fh*

INT 2Fh Function 122Ah DOS 3.3+

SET FASTOPEN ENTRY POINT

Specify the address(es) of the handlers for the FASTOPEN filename cache.

Call with:

AX	122Ah
BX	entry point to set (0001h or 0002h)
DS:SI	pointer to FASTOPEN entry point (entry point not set if SI=FFFFh for DOS 4+)

Returns:

CF set if specified entry point already set

Notes:

- The entry point in BX is ignored under DOS 3.30.
- Both entry points are set to the same handler by DOS 4.01 FASTOPEN.

DOS 3.30 FASTOPEN is called with:

AL 01h nknown

CX	*seems to be an offset*
DI	*seems to be an offset*
SI	offset in DOS CS of filename

AL 02h unknown

AL 03h open file

SI offset in DOS CS of filename
AL 04h unknown
AH subfunction (00h,01h,02h)
ES:DI unknown pointer
CX unknown (subfunctions 01h and 02h only)

Returns:
CF set on error or not installed

Note:
function 03h calls function 01h first

PCDOS 4.01 FASTOPEN is additionally called with:
AL 04h
AH 03h unknown
AL 05h unknown
AL 0Bh unknown
AL 0Ch unknown
AL 0Dh unknown
AL 0Eh unknown
AL 0Fh unknown
AL 10h unknown

INT 2Fh Function 122Bh DOS 3.3+

IOCTL

Execute an I/O Control function from within network redirector.

Call with:
AX 122Bh
BP 44xxh
SS set to DOS CS
additional registers as appropriate for *INT 21/AX=44xxh*

Returns:
as for *INT 21/AH=44h*

Note:
This function is equivalent to INT 21/AH=44h, but may only be called when already inside a DOS function call.

See Also: *INT 21/AH=44h*

INT 2Fh Function 122Ch DOS 3.3+

GET DEVICE CHAIN

Return a pointer to the device driver chain (omitting the NUL device).

Call with:
AX 122Ch

Returns:
BX:AX pointer to header of second device driver (NUL is first) in driver chain

See Also: INT 21/AH=52h

INT 2Fh Function 122Dh DOS 3.3+

GET EXTENDED ERROR CODE

Return the current extended error code.

Call with:
AX 122Dh

Returns:
AX current extended error code

See Also: INT 2F/AX=1222h, *INT 21/AH=59h*

INT 2Fh Function 122Eh DOS 4.0+

GET OR SET ERROR TABLE ADDRESSES

Specify or determine the locations of various tables used to convert error numbers into error messages.

Call with:

AX 122Eh
DL subfunction
00h get standard DOS error table (errors 00h-12h,50h-5Bh)
Returns: ES:DI pointer to error table
01h set standard DOS error table
ES:DI pointer to error table
02h get parameter error table (errors 00h-0Ah)
Returns: ES:DI pointer to error table
03h set parameter error table
ES:DI pointer to error table
04h get critical/SHARE error table (errors 13h-2Bh)
Returns: ES:DI pointer to error table
05h set critical/SHARE error table
ES:DI pointer to error table
06h get unknown error table
Returns: ES:DI pointer to error table
07h set unknown error table
ES:DI pointer to error table
08h get unknown error table
Returns: ES:DI pointer to error table
09h set unknown error table
ES:DI pointer to error table

Format of error table:

Offset	Size	Description
00h	BYTE	FFh
01h	*2 BYTEs*	*04h,00h (may be DOS version)*
03h	BYTE	number of error headers following
04h	2N WORDs	table of all error headers for table

Offset	Size	Description
00h	WORD	error message number

02h	WORD	offset of error message from start of header error messages are count byte followed by msg

See Also: *INT 21/AH=59h*

INT 2Fh Function 122Fh DOS 4.0+

SET UNKNOWN

The purpose of this function is not known.

Call with:

AX	122Fh
DX	*unknown*

INT 2Fh Function 13h DOS 3.3+

SET DISK INTERRUPT HANDLER

Specify the address of the handler for most DOS disk access, and return the old handler's address.

Call with:

AH	13h
DS:DX	pointer to interrupt handler disk driver calls on read/write
ES:BX	address to restore INT 13 to on system halt (exit from root shell)

Returns:

DS:DX from previous invocation of this function
ES:BX from previous invocation of this function

Notes:

- Most DOS 3.3+ disk access is performed via the vector set in DS:DX, although a few functions are still invoked via an INT 13 instruction.
- This can a dangerous security loophole for any virus-monitoring software which does not trap this call. At least two viruses are known to use it to get the original ROM entry point.

INT 2Fh Function 1400h — DOS 3.3+

NLSFUNC.COM - INSTALLATION CHECK

Determine whether NLSFUNC has been loaded.

Called with:

AX 1400h

Returns:

AL 00h not installed, OK to install
01h not installed, not OK
FFh installed

Note:

This function is called by the DOS v3.3+ kernel.

INT 2Fh Function 1401h — DOS 3.3+

NLSFUNC.COM - CHANGE CODE PAGE

Select a new code page as the default.

Called with:

AX 1401h
DS:SI pointer to internal code page structure (see below)
BX new code page
DX *country code*

Returns:

AL status
00h successful
else DOS error code

Note:

This function is called by the DOS v3.3+ kernel.

Format of DOS 3.30 internal code page structure:

Offset	Size	Description
00h	*8 BYTEs*	*unknown*
08h	64 BYTEs	name of country information file
48h	WORD	system code page
4Ah	WORD	number of supported subfunctions
4Ch	5 BYTEs	data to return for INT 21/AX=6502h
51h	5 BYTEs	data to return for INT 21/AX=6504h
56h	5 BYTEs	data to return for INT 21/AX=6505h
5Bh	5 BYTEs	data to return for INT 21/AX=6506h
60h	41 BYTEs	data to return for INT 21/AX=6501h

See Also: *INT 21/AH=66h*

INT 2Fh Function 1402h DOS 3.3+

NLSFUNC.COM - GET COUNTRY INFO

Get country-specific information for a country or code page other than the default.

Called with:

AX	1402h
BP	subfunction (same as AL for INT 21/AH=65h)
BX	code page
DX	country code
DS:SI	pointer to internal code page structure (see INT 2F/AX=1401h)
ES:DI	pointer to user buffer
CX	size of user buffer

Returns:

AL status
00h successful
else DOS error code

Notes:

- This function is called by the DOS v3.3+ kernel on INT 21/AH=65h.
- The code page structure is apparently only needed for the COUNTRY.SYS pathname.

See Also: INT 2F/AX=1403h,INT 2F/AX=1404h, INT 21/AH=65h

INT 2Fh Function 1403h DOS 3.3+

NLSFUNC.COM - SET COUNTRY INFO

Select a new country code as the default.

Called with:

AX	1403h
DS:SI	pointer to internal code page structure (see INT 2F/AX=1401h)
BX	code page
DX	country code

Returns:

AL status

Note:

This function is called by the DOS v3.3+ kernel on INT 21/AH=38h.

See Also: INT 2F/AX=1402h, INT 2F/AX=1404h, *INT 21/AH=38h*

INT 2Fh Function 1404h DOS 3.3+

NLSFUNC.COM - GET COUNTRY INFO

Return country-specific information for a country other than the current default.

Called with:

AX	1404h
BX	code page
DX	country code
DS:SI	pointer to internal code page structure (see INT 2F/AX=1401h)
ES:DI	pointer to user buffer

Returns:

AL status

Notes:

- This function is called by the DOS v3.3+ kernel on INT 21/AH=38h.
- The code page structure is apparently only needed for the COUNTRY.SYS pathname.

See Also: INT 2F/AX=1402h, INT 2F/AX=1403h, *INT 21/AH=38h*

INT 2Fh Function 1500h — CDROM extensions

MICROSOFT CD-ROM EXTENSIONS (MSCDEX) INSTALLATION CHECK

Although documented, this function is included because it conflicts with the GRAPHICS.COM installation check for DOS 4+.

Call with:

AX 1500h
BX 0000h

Returns:

BX number of CDROM drive letters used
CX starting drive letter (0=A:)

Note:

This installation check DOES NOT follow the format used by other software.

INT 2Fh Function 1500h — DOS 4+

GRAPHICS.COM - INSTALLATION CHECK

Determine whether GRAPHICS has been loaded.

Call with:

AX 1500h

Returns:

AX FFFFh
ES:DI *pointer to unknown information (perhaps graphics data)*

INT 2Fh Function 1900h — DOS 4.x only

SHELLB.COM - INSTALLATION CHECK

Determine whether SHELLB has been loaded.

Call with:

AX 1900h

Returns:

AL 00h not installed
FFh installed

INT 2Fh Function 1901h DOS 4.x only

SHELLB.COM - SHELLC.EXE INTERFACE

Inform SHELLB of SHELLC's address, and return the location of a workspace for SHELLC.

Call with:

AX 1901h
BL 00h if SHELLC transient
01h if SHELLC resident
DS:DX pointer to far call entry point for resident SHELLC.EXE

Returns:

ES:DI pointer to SHELLC.EXE workspace within SHELLB.COM

Note:

SHELLB.COM and SHELLC.EXE are parts of the DOS 4.x shell.

INT 2Fh Function 1902h DOS 4.x only

SHELLB.COM - COMMAND.COM INTERFACE

Get the next line which COMMAND.COM should execute in preference to reading a command from the current batch file.

Call with:

AX 1902h
ES:DI pointer to ASCIZ full filename of current batch file, with at least the final filename element uppercased
DS:DX pointer to buffer for results

Returns:

AL 00h failed, either
(a) the final filename element quoted at ES:DI does not match the identity of shell batch file quoted as the parameter of the most recent call of the SHELLB command, or

(b) no more Program Start Commands are available.

AL= FFh success, then:

memory at DS:[DX+1] onwards filled as:

DX+1:	BYTE	count of bytes of PSC
DX+2:	N BYTEs	Program Start Command text
	BYTE	0Dh terminator

Notes:

- As long as SHELLB provides Program Start Commands from its workspace, the current batch file does not advance.
- The final PSC of a sequence is finished with a GOTO COMMON, which causes a loop back in the batch file which called SHELLC so as to execute SHELLC again.
- The PSCs are planted in SHELLB workspace by SHELLC, the user menu interface.
- The check on batch file name permits PSCs to CALL nested batch files while PSCs are still stacked up for subsequent execution.

INT 2Fh Function 1903h DOS 4.x only

SHELLB.COM - COMMAND.COM interface

Determine whether Program Start Command is attempting to re-execute the current batch file.

Call with:

AX 1903h

ES:DI pointer to ASCIZ batch file name as for AX=1902h

Returns:

AL FFh if quoted batch filename matches last SHELLB parameter

AL 00h if it does not

INT 2Fh Function 1904h DOS 4.x only

SHELLB.COM - SHELLB.COM transient to TSR interface

Determine the name of the batch file from which the DOS Shell is executing.

Call with:

AX 1904h

Returns:

ES:DI pointer to name of current shell batch file:
- WORD number of bytes of name following
- BYTEs (8 max) uppercase name of shell batch file

INT 2Fh Function 1A00h DOS 4+

ANSI.SYS - INSTALLATION CHECK

Determine whether ANSI.SYS is present.

Call with:

AX 1A00h

Returns:

AL FFh if installed

INT 2Fh Function 1A01h DOS 4+

ANSI.SYS - GET/SET DISPLAY INFORMATION

This is presumably the DOS IOCTL interface to ANSI.SYS.

Call with:

AX 1A01h
CL 7Fh for GET
5Fh for SET
DS:DX pointer to parm block as for INT 21,AX=440Ch,CX=037Fh/035Fh respectively

Returns:

CF set on error
- AX error code (many non-standard)

CF clear if successful
- AX destroyed

See Also: *INT 21/AX=440Ch,* INT 2F/AX=1A02h

INT 2Fh Function 1A02h DOS 4+

ANSI.SYS - MISC REQUESTS

Get or set miscellaneous ANSI.SYS flags.

Call with:

AX	1A02h
DS:DX	pointer to parameter block (see below)

Format of parameter block:

Offset	Size	Description
00h	BYTE	subfunction 00h set/reset interlock 01h get /L flag
01h	BYTE	interlock state 00h=reset, 01h=set This interlock prevents some of the ANSI.SYS post-processing in its hook onto INT 10, AH=00h mode set
02h	BYTE	(returned) 00h if /L not in effect 01h if /L in effect

See Also: INT 2F/AX=1A01h

INT 2Fh Function 1B00h DOS 4+

XMA2EMS.SYS - INSTALLATION CHECK

Determine whether XMA2EMS.SYS has been loaded.

Call with:

AX	1B00h

Returns:

AL FFh if installed

Notes:

- The XMA2EMS.SYS extension is only installed if DOS has page frames to hide.

- This extension hooks onto INT 67/AH=58h and returns from that call data which excludes the physical pages being used by DOS.

See Also: INT 2F/AX=1B01h

INT 2Fh Function 1B01h DOS 4+

XMA2EMS.SYS - GET HIDDEN FRAME INFO

Determine information which XMA2EMS hides from regular EMS function calls.

Call with:

AX 1B01h
DI hidden physical page number

Returns:

AX FFFFh if failed (no such hidden page)
AX 0000h if OK, then
ES segment of page frame
DI physical page number

Note:

The returned data corresponds to the data edited out of the INT 67/AH=58h call by XMA2EMS.

See Also: INT 2F/AX=1BFFh

INT 2Fh Function 1BFFh DOS 4+

XMA2EMS.SYS - UNKNOWN

The purpose of this function is not known, but it appears likely to be the same as INT 2F/AX=1B01h.

Call with:

AX 1BFFh
DI unknown

Returns:
AH *unknown*
ES:DI *pointer to unknown*

Note:
This function is called by FASTOPEN.

INT 2Fh Function 4001h — OS/2 compatibility box

SWITCH DOS TO BACKGROUND

Place the OS/2 compatibility box in the background.

Call with:
AX 4001h

See Also: INT 2F/AX=4002h

INT 2Fh Function 4002h — OS/2 compatibility box

SWITCH DOS TO FOREGROUND

Place the OS/2 compatibility box in the foreground.

Call with:
AX 4002h

See Also: INT 2F/AX=4001h

INT 2Fh Function AD00h — DOS 3.3+

DISPLAY.SYS - INSTALLATION CHECK

Determine whether DISPLAY.SYS is present.

Call with:
AX AD00h

Returns:

AL FFh if installed
BX *unknown (0100h in MS-DOS 3.30, PCDOS 4.01)*

INT 2Fh Function AD01h DOS 3.3+

DISPLAY.SYS - SET UNKNOWN VALUE

The purpose of the value set by this call is not known.

Call with:

AX AD01h
BX *unknown*

Returns:

CF set on error
additional return values (if any) unknown

INT 2Fh Function AD02h DOS 3.3+

DISPLAY.SYS - GET UNKNOWN VALUE

The purpose of the value returned by this call is not known.

Call with:

AX AD02h

Returns:

BX *unknown (value set with AX=AD01h)*

INT 2Fh Function AD03h DOS 3.3+

DISPLAY.SYS - GET UNKNOWN INFORMATION

The purpose and format of the data returned by this call are not known.

Call with:

AX AD03h

ES:DI pointer to user buffer
CX size of buffer

Returns:
CF set if buffer too small
CF clear on success

INT 2Fh Function AD04h DOS 4+

DISPLAY.SYS - UNKNOWN

The purpose of this function is not known.

Call with:
AX AD04h
additional arguments (if any) unknown

Returns:
unknown

INT 2Fh Function AD10h DOS 4+

DISPLAY.SYS internal - INSTALLATION CHECK

Determine whether DISPLAY.SYS has been loaded.

Call with:
AX AD10h
additional arguments (if any) unknown

Returns:
AX FFFFh
BX unknown (0100h in PCDOS 4.01)

INT 2Fh Function AD40h — DOS 4+

UNKNOWN

The purpose of this function is not known.

Call with:

AX AD40h
DX unknown
additional arguments (if any) unknown

Returns:

unknown

Note:

This function is called by the DOS 4.01 PRINT.COM.

INT 2Fh Function AD80h — DOS 3.3+

KEYB.COM - INSTALLATION CHECK

Determine whether KEYB.COM has been loaded.

Call with:

AX AD80h

Returns:

AL FFh if installed
BX unknown (0100h in MSDOS 3.30 and PCDOS 4.01)
ES:DI pointer to internal data (see below)

Format of KEYB internal data:

Offset	Size	Description
00h	DWORD	original INT 09
04h	DWORD	original INT 2F
08h	*6 BYTEs*	*unknown*
0Eh	WORD	flags
10h	*BYTE*	*unknown*
11h	*BYTE*	*unknown*

12h	*4 BYTEs*	*unknown*
16h	2 BYTEs	country ID letters
18h	WORD	current code page

DOS 3.3

1Ah	*WORD*	*pointer to first item in list of code page tables*
1Ch	*WORD*	*pointer to an item in list of code page tables*
1Eh	*2 BYTEs*	*unknown*
20h	WORD	pointer to key translation data
22h	WORD	pointer to last item in code page table list (see below)
24h	*9 BYTEs*	*unknown*

DOS 4.01

1Ah	*2 BYTEs*	*unknown*
1Ch	*WORD*	*pointer to first item in list of code page tables*
1Eh	*WORD*	*pointer to an item in list of code page tables*
20h	*2 BYTEs*	*unknown*
22h	WORD	pointer to key translation data
24h	WORD	pointer to last item in code page table list (see below)
26h	*9 BYTEs*	*unknown*

Format of code page table list entries:

Offset	Size	Description
00h	WORD	pointer to next item, FFFFh if last
02h	WORD	code page
04h	*2 BYTEs*	*unknown*

Format of translation data:

Offset	Size	Description
00h	WORD	size of data in bytes, including this word
02h	*N-2 BYTEs*	*unknown*

INT 2Fh Function AD81h — DOS 3.3+

KEYB.COM - SET KEYBOARD CODE PAGE

Select a new code page for use by the keyboard driver.

Call with:

AX	AD81h
BX	code page

Returns:
CF set on error
AX 0001h (code page not available)
CF clear if successful

Note:
This function is called by DISPLAY.SYS.

See Also: INT 2F/AX=AD82h

INT 2Fh Function AD82h DOS 3.3+

KEYB.COM - SET KEYBOARD MAPPING

Specify whether the keyboard driver should use the standard or the foreign key mappings.

Call with:

AX	AD82h
BL	00h US keyboard (Control-Alt-F1)
	FFh foreign keyboard (Control-Alt-F2)

Returns:
CF set on error (BL not 00h or FFh)
CF clear if successful

See Also: INT 2F/AX=AD81h

INT 2Fh Function AE00h DOS 3.3+

INSTALLABLE COMMAND - INSTALLATION CHECK

Determine whether a command is a TSR extension to COMMAND.COM's internal command set.

Called with:

AX	AE00h
DX	FFFFh
DS:BX	pointer to command line
CH	first call: FFh
	second call: 0

Returns:

AL FFh if this command is a TSR extension to COMMAND.COM
00h if the command should be executed as usual

Notes:

- This call provides a mechanism for TSRs to install permanent extensions to the command repertoire of COMMAND.COM. It appears that COMMAND.COM makes this call before executing the current command line, and does not execute it itself if the return is FFh.
- APPEND hooks this call, to allow subsequent APPEND commands to execute without reloading APPEND from disk.

Format of command line:

Offset	Size	Description
00h	BYTE	max length of command line, as in *INT 21/AH=0Ah*
01h	BYTE	count of bytes to follow
	N BYTEs	command line text, terminated by 0Dh

INT 2Fh Function AE01h — DOS 3.3+

INSTALLABLE COMMAND - EXECUTE

Execute a TSR extension to COMMAND.COM's set of internal commands. The extension may resolve to an existing internal command.

Called with:

AX AE01h
DX FFFFh
DS:SI pointer to buffer

Returns:

buffer at DS:SI filled with a length byte followed by the uppercase internal command to execute (if length not 0)

Notes:

- This call requests execution of the command which a previous call to AX=AE00h indicated was resident.
- APPEND hooks this call.

- If the buffer is returned with a nonempty string, COMMAND.COM attempts to execute it as an internal command.

INT 2Fh Function B000h — DOS 3.3+

GRAFTABL.COM - INSTALLATION CHECK

Determine whether GRAFTABL has been loaded.

Call with:

AX	B000h

Returns:

AL	00h not installed, OK to install
	01h not installed, not OK to install
	FFh installed

Note:

This function is called by DISPLAY.SYS.

INT 2Fh Function B001h — DOS 3.3+

GRAFTABL.COM - GET UNKNOWN INFORMATION

Return miscellaneous information about the resident GRAFTABL.

Call with:

AX	B001h
DS:DX	pointer to 4-byte buffer (see below)

Returns:

buffer filled

AL	FFh

Format of buffer:

Offset	Size	Description
00h	*WORD*	*unknown (PCDOS 3.30/4.01 fill in 0130h, MSDOS 3.30 fills in 0030h)*
02h	WORD	CS of resident code

INT 2Fh Function B700h DOS 3.3+

APPEND - INSTALLATION CHECK
Determine whether APPEND has been loaded.

Call with:
AX B700h

Returns:
AL 00h not installed
FFh if installed

Note:
MSDOS 3.30 APPEND refuses to install itself when run inside TopView or a TopView-compatible environment

INT 2Fh Function B701h DOS 3.3+

APPEND - UNKNOWN

The purpose of this function is not known.

Call with:
AX B701h
additional arguments (if any) unknown

Note:
MSDOS 3.30 APPEND displays "Incorrect APPEND Version" and aborts the caller.

INT 2Fh Function B702h DOS 3.3+

APPEND - VERSION CHECK

Determine which version of APPEND has been loaded.

Call with:
AX B702h

Returns:

AX	FFFFh if not DOS 4.0 APPEND
AL	major version number
AH	minor version number, otherwise

See Also: INT 2F/AX=B710h

INT 2Fh Function B703h DOS 3.3

APPEND - HOOK INT 21

Place a handler on the INT 21h call chain after APPEND.

Call with:

AX	B703h
ES:DI	pointer to INT 21h handler APPEND should chain to

Returns:

ES:DI	pointer to APPEND's INT 21h handler

Note:
Each invocation of this function toggles a flag which APPEND uses to determine whether to chain to the user handler or the original INT 21.

INT 2Fh Function B704h DOS 3.3+

APPEND - GET APPEND PATH

Return the current APPEND path.

Call with:

AX	B704h

Returns:

ES:DI	pointer to active APPEND path (128 bytes max)

INT 2Fh Function B710h DOS 3.3+

APPEND - GET VERSION INFO

Appears to return various information in addition to the version of APPEND which was loaded.

Call with:

AX B710h

Returns:

AX *unknown*
BX *unknown (0000h in MSDOS 3.30)*
CX *unknown (0000h in MSDOS 3.30)*
DL major version
DH minor version

See Also: INT 2F/AX=B702h

INT 2Fh Function B800h network

INSTALLATION CHECK

Determine whether a network is installed.

Call with:

AX B800h

Returns:

AL 00h not installed
nonzero installed
BX installed component flags (test in this order!)
bit 6 server
bit 2 messenger
bit 7 receiver
bit 3 redirector

INT 2Fh Function B803h network

GET CURRENT POST HANDLER ADDRESS

This function is used in conjunction with INT 2F/AX=B804h to hook into the network event post routine.

Call with:

AX B803h

Returns:

ES:BX post address

See Also: INT 2F/AX=B804h, INT 2F/AX=B903h

INT 2Fh Function B804h network

SET NEW POST HANDLER ADDRESS

This function is used in conjunction with INT 2F/AX=B803h to hook into the network event post routine.

Call with:

AX B804h
ES:BX new FAR post handler address

Note:

The specified handler is called on any network event. Two events are defined: message received and critical network error.

Values post routine is called with:

AX = 0000h single block message
    DS:SI - ASCIZ originator name
    DS:DI - ASCIZ destination name
    ES:BX - text header (see below)
AX = 0001h start multiple message block
    CX = block group ID
    DS:SI - ASCIZ originator name
    DS:DI - ASCIZ destination name

AX = 0002h multiple block text
CX = block group ID
ES:BX - text header (see below)
AX = 0003h end multiple block message
CX = block group ID
AX = 0004h message aborted due to error
CX = block group ID
AX = 0101h server received badly formatted network request
Returns: AX = FFFFh (PC LAN will process error)
AX = 0102h unexpected network error
ES:BX - NCB (see INT 5C)
AX = 0103h server received INT 24 error
other registers as for INT 24, except AH is in BH
Returns: as below, but only 0000h and FFFFh allowed
Post routine returns:
AX=response code
0000h user post routine processed message
0001h PC LAN will process message, but message window not displayed
FFFFh PC LAN will process message

Format of text header:

Offset	Size	Description
00h	WORD	length of text (maximum 512 bytes)
02h	N BYTEs	text of message

Note:

All CRLF sequences in the message text are replaced by ASCII 14h.

SeeAlso: INT 2F/AX=B803h, INT 2F/AX=B904h

INT 2Fh Function B807h network

GET NetBIOS NAME NUMBER OF MACHINE NAME

Return the network machine number.

Call with:

AX B807h

Returns:
CH NetBIOS name number of the machine name

See Also: INT 21/AX=5E00h

INT 2Fh Function B808h network

UNKNOWN

The purpose of this function is not known.

Call with:
AX B808h
additional arguments (if any) unknown

Returns:
unknown

INT 2Fh Function B809h network

VERSION CHECK

Determine which version of the network software has been installed.

Call with:
AX B809h

Returns:
AH major version
AL minor version

INT 2Fh Function B900h PC Network RECEIVER.COM

INSTALLATION CHECK

Determine whether the PC Network receiver module has been loaded.

Call with:
AX B900h

Returns:

AL 00h if not installed
FFh if installed

INT 2Fh Function B901h — PC Network RECEIVER.COM

GET RECEIVER.COM INT 2Fh HANDLER ADDRESS

Return the entry point for the RECEIVER.COM INT 2Fh handler, allowing more efficient execution by bypassing any other handlers which have hooked INT 2Fh since RECEIVER.COM was installed.

Call with:

AX B901h

Returns:

AL *unknown*
ES:BX pointer to RECEIVER.COM INT 2Fh handler

INT 2Fh Function B903h — PC Network RECEIVER.COM

GET RECEIVER.COM POST ADDRESS

This function is used in conjunction with INT 2F/AX=B904h to hook into the network event post routine.

Call with:

AX B903h

Returns:

ES:BX pointer to POST handler

See Also: INT 2F/AX=B803h, INT 2F/AX=B904h

INT 2Fh Function B904h — PC Network RECEIVER.COM

SET RECEIVER.COM POST ADDRESS

This function is used in conjunction with INT 2F/AX=B903h to hook into the network event post routine.

Call with:

AX	B904h
ES:BX	pointer to new POST handler

See Also: INT 2F/AX=B804h, INT 2F/AX=B903h

INT 2Fh Function B905h — PC Network RECEIVER.COM

GET FILENAME

Return two filenames used internally by RECEIVER.COM.

Call with:

AX	B905h
DS:BX	pointer to 128-byte buffer for filename 1
DS:DX	pointer to 128-byte buffer for filename 2

Returns:

buffers filled from RECEIVER.COM internal buffers

Note:

The use of the filenames is unknown, but one appears to be for storing messages.

See Also: INT 2F/AX=B906h

INT 2Fh Function B906h — PC Network RECEIVER.COM

SET FILENAME

Specify the filenames which RECEIVER.COM uses internally.

Call with:

AX	B906h
DS:BX	pointer to 128-byte buffer for filename 1
DS:DX	pointer to 128-byte buffer for filename 2

Returns:

RECEIVER.COM internal buffers filled from user buffers

Note:

The use of the filenames is unknown, but one appears to be for storing messages.

See Also: INT 2F/AX=B905h

INT 2Fh Function B908h — PC Network RECEIVER.COM

UNLINK KEYBOARD HANDLER

Remove the INT 09h handler immediately following RECEIVER.COM in the INT 09h chain.

Call with:

AX	B908h
ES:BX	pointer to INT 09 handler RECEIVER should call after it finishes INT 09

Note:

This call replaces the address to which RECEIVER.COM chains on an INT 09 without preserving the original value. This allows a prior handler to unlink, but does not allow a new handler to be added such that RECEIVER gets the INT 09 first.

INT 2Fh Function BF00h — PC LAN PROGRAM REDIRIFS.EXE

INSTALLATION CHECK

Determine whether the PC LAN Program Installable File System module has been loaded.

Call with:

AX BF00h

Returns:

AL FFh if installed

INT 2Fh Function BF01h — PC LAN PROGRAM REDIRIFS.EXE

UNKNOWN

The purpose of this function is not known.

Call with:

AX BF01h

additional arguments (if any) unknown

Returns:

unknown

INT 2Fh Function BF80h — PC LAN PROGRAM REDIR.SYS

SET REDIRIFS ENTRY POINT

Specify the address of an Installable File System handler for the PC LAN Program redirector.

Call with:

AX BF80h

ES:DI pointer to FAR entry point to IFS handler in REDIRIFS

Returns:

AL FFh if installed
ES:DI pointer to internal workspace

Note:

After executing this function, all future IFS calls to REDIR.SYS are passed to the specified entry point.

INT 30h DOS 1+

(NOT A VECTOR!) FAR JMP INSTRUCTION FOR CP/M-STYLE CALLS

For CP/M compatibility, a program may invoke DOS function calls by loading CL with the function number and performing a near call to offset 5 in the program's PSP. That location contains a FAR jump instruction which points at the absolute address 000C0h, which is this vector. This vector contains a FAR jump instruction to the CP/M-compatible entry point for the INT 21h handler.

Note:

The jump address in the PSP is two bytes too low (it points into the middle of INT 2Fh) under DOS 2 and up, if the PSP was created by EXEC (INT 21/AH=48h).

See Also: INT 21/AH=26h

INT 31h DOS 1+

OVERWRITTEN BY CP/M JUMP INSTRUCTION IN INT 30h

The first byte of this vector contains the end of the FAR JMP instruction stored in INT 30h.

Appendix B

Annotated Bibliography

Jon K. Adams, "Reading and Writing the DOS Environment," *Computer Language*, April 1989, pp. 45-51.
Presents Turbo Pascal code which uses undocumented fields in the PSP to locate COMMAND.COM.

Phillip M. Adams and Clovis L. Tondo, *Writing DOS Device Drivers in C*, Englewood Cliffs NJ: Prentice Hall, 1990, 385 pp.
This book presents a fine explanation of writing device drivers in C rather than in assembly language.

Nancy Andrews, "Moving Toward an Industry Standard for Developing TSRs," *Microsoft Systems Journal*, December 1986, pp. 7-12.
This article from a Microsoft publication notes that undocumented DOS features such as the InDOS flag are "critical for TSRs to work consistently."

Douglas Boling, "Background Copying Without OS/2," *PC Magazine*, 17 January 1989, pp. 289-315.
Presents a DOS multitasking TSR which copies files in the background; includes discussion of INT 28h and of Get/Set PSP (including a fix for a DOS 2.x problem with the PSP calls).

Douglas Boling and Jeff Prosise, "Give Yourself a Smart DOS Command Line with ALIAS," *PC Magazine*, 26 December 1989, pp. 253-268.
Discusses a variety of undocumented DOS topics, including INT 21h Function 51h (Get PSP) in DOS 2.x, finding COMMAND.COM's PSP, and changes brought about by the INSTALL= statement in DOS 4+.

"Dr. Bob," "Undocumented 34H Call," *Microsoft Systems Journal*, September 1987, pp. 77-78.
Another rare Microsoft description of undocumented DOS.

Ken W. Christopher, Jr., Barry A. Feigenbaum, Shon O. Saliga, *Developing Applications Using DOS*, New York NY: John Wiley & Sons, 1990, 573 pp.
This book by three IBM employees who were "the lead engineers directing the entire development of the DOS 4.0 system" describes many undocumented DOS features, including for example INT 21h Function 5Dh.

Terry Dettmann and Jim Kyle, *DOS Programmer's Reference*, Second Edition, Carmel IN: Que Corporation, 1989, 892 pp.
Contains extensive reference material on undocumented DOS, and a fifty-page appendix by Jim Kyle.

Ray Duncan, *Advanced MS-DOS Programming*, Second Edition, Redmond WA: Microsoft Press, 1988, 669 pp.
This is the bible of DOS programming, with examples in assembler and C.

Ray Duncan (editor), *The MS-DOS Encyclopedia*, Redmond WA: Microsoft Press, 1988, 1570 pp.
Part C of this mammoth book ("Customizing MS-DOS") is particularly good, containing chapters on TSRs, exception handlers, hardware interrupt handlers, DOS filters, and installable device drivers. Includes Richard Wilton's definitive piece on TSR programming.

Earl F. Glynn, "Getting a Good Look at How DOS Allocates Your Memory Blocks," *PC Magazine*, 12 June 1990, pp. 329-344.
A detailed look at building a MCB walker with Turbo Pascal; also discusses the MCB chain in the DOS compatibility box of OS/2, and presents an unsolved "memory enigma."

Daniel E. Greenberg, "Reentering the DOS Shell," *Programmer's Journal*, May-June 1990, pp. 28-36.
The definitive examination of the COMMAND.COM backdoor, INT 2Eh.

Robert L. Hummel, "How the DOS CLS Command Handles Various Displays," *PC Magazine*, 11 October 1988, pp. 341-344.
"Every time you use the CLS command, DOS sifts through a bewildering array of information. Here's a peek behind the scenes of how COMMAND.COM clears the screen"; includes a discussion of INT 29h and the SPECL bit in the device driver header.

Intel Corporation, "Undocumented iAPX 286 Test Instruction," 1987.
Intel's 15-page document on the undocumented 80286 LOADALL instruction.

Rahner James, "Undocumented DOS," *Dr. Dobb's Journal*, June 1989, pp. 26-34.
Discusses INT 21h Function 52h, the DPB, SFT (referred to as the "Open File Table"), CDS (referred to as the "Drive Path Table"), and sector buffers. See also the letter by Michael Cook, "Delving into Drive Paths," in Dr. Dobb's Journal, February 1990, pp. 12-14.

Jim Kyle, "The Reserved DOS Functions," in Dettmann and Kyle, *DOS Programmer's Reference*, pp. 805-856.
Presents a program, with source code in Turbo Pascal, for examining the "Configuration Variable Table" (CVT) in DOS 2.x, 3.x, and 4.x. The CVT is what in Undocumented DOS we refer to as the List of Lists.

Robert S. Lai and The Waite Group, *Writing MS-DOS Device Drivers*, Reading MA: Addison-Wesley, 1987, 466 pp.
This is the standard work on DOS device drivers; all examples are in assembler.

Michael Mefford, "Choose CONFIG.SYS Options at Boot," *PC Magazine*, 29 November 1988, pp. 323-34.
Contains a discussion of the undocumented DOS CONFIG.SYS buffer.

Michael Mefford, "Running Programs Painlessly," *PC Magazine*, 16 February 1988, pp. 321-336.
Contains a discussion of the problems with using INT 2Eh (though the author's assertion that INT 2Eh "will not execute batch files nor work from within a batch file" appears not to be true in DOS 3.x).

Raymond J. Michels, "Undocumented DOS Internals," *Programmer's Journal* 7.2 (1989), pp. 32-37.
Briefly discusses the following areas of undocumented DOS: PSP, environment, MCBs, the SFT, DPBs, and the DOS variables table.

Ted Mirecki, "DOS Memory Control," PC Tech Journal, October 1987, p. 45.
A brief discussion of the layout of MCBs, from the popular "Tech Notebook" in a now-defunct magazine.

Ted Mirecki, "Function 32H in DOS," *PC Tech Journal*, February 1989, pp. 129-133.
Describes the structure of the DPB.

Ted Mirecki, "More Handles for New Applications" and "More Handles for Old Applications," *PC Tech Journal*, April 1988, pp. 161-165.
Describes the file handle table within a process's PSP.

Charles Petzold, "Widening the Path," *PC Magazine*, 28 April 1987, pp. 341-348.
Discusses "the undocumented (and strange)" INT 2Eh.

Jeff Prosise, "How Device Drivers Work," *PC Magazine*, 28 November 1989, pp. 379-380.
Discusses finding the device chain via INT 21h Function 52h.

Jeff Prosise, "The Inner Life of a TSR," *PC Magazine*, August 1990, pp. 467-472.
General discussion of TSRs, including the InDOS flag, INT 28h, critical error flag, and Get/Set PSP.

Jeff Prosise, "Instant Access to Directories," *PC Magazine*, 14 April 1987, pp. 313-334.
Excellent discussion of TSR programming; discusses the InDOS flag, INT 28h (including the need, not only to hook INT 28h, but also to periodically invoke INT 28h as well), the DOS stacks, DTA, critical errors, and hooking the BIOS disk interrupt.

Jeff Prosise, "Teaching a TSR New Tricks," *PC Magazine*, 12 June 1990, p. 384.
A brief discussion of the active PSP.

Robin Raskin, Charles Petzold, and Stephen Randy Davis, "Taking Up Residence," *PC Magazine*, 25 November 1986, pp. 163-193.
A detailed look at the "TSR blues."

Tony Rizzo, "MS-DOS CD-ROM Extensions: A Standard PC Access Method," *Microsoft Systems Journal*, September 1987, pp. 54-60.
Describes the Microsoft CD-ROM Extensions (MSCDEX), including its implementation using the DOS network redirector.

Arthur Rothstein, "Walking the OS/2 Device Chain," *Dr. Dobb's Journal*, October 1990, p. 30.
An interesting contrast to walking the device chain under MS-DOS.

Herbert Schildt, "Supercharging TSR's," *Born to Code in C*, Berkeley CA: Osborne McGraw-Hill, 1989, pp. 139-201.
Chapter 3 of this book presents a useful TSR skeleton in Turbo C, which uses the usual assortment of undocumented DOS functions for building robust TSRs. This is an update to the author's earlier discussion of TSRs in C: Power User's Guide *(Berkeley CA: Osborne McGraw-Hill, 1988, Chapter 3, pp. 91-128), which had asserted that the interrupt service routines in a TSR "cannot use any DOS functions." The later book's TSRs are "supercharged" in the sense that they can make DOS calls, though only because they employ undocumented DOS.*

Barry Simon, "Providing Program Access to the Real DOS Environment," *PC Magazine*, 28 November 1989, pp. 309-314.
Discusses the three (count 'em) DOS environments—the real or active environment, the program environment, and the root environment—and shows how to access the first of these. This provides a useful contrast to our own techniques in chapter 6, which emphasize accessing what this article calls the "root" environment.

Al Stevens, *Turbo C: Memory-Resident Utilities, Screen I/O and Programming Techniques*, Portland OR: MIS Press, 1987, 315 pp.
Chapter 11 ("Memory-Resident Programs") and Chapter 12 ("Building Turbo C Memory-Resident Programs") present a TSR skeleton, using undocumented DOS. The author's later Extending Turbo C Professional *(Portland OR: MIS Press, 1989, 418 pp.) contains an improved TSR skeleton, plus a TSR overlay manager.*

John Switzer, "Closing DOS's Backdoor," *Dr. Dobb's Journal*, October 1990, pp. 42-48.
Explains a bizarre quirk of the DOS Delete FCB function (INT 21h Function 13h), with details on the INT 30h and INT 31h alternate function dispatchers.

Giles Todd, "Installing MS-DOS Device Drivers from the Command Line," *.EXE*, August 1989, pp. 16-20.
Presents assembly-language code for loading device drivers under DOS 3.30; includes discussion of INT 21h Function 52h.

Thomas A. Wadlow, *Memory Resident Programming on the IBM PC*, Reading MA: Addison-Wesley, 1987, 413 pp.
This book is well written and well organized, but the author refrains from using undocumented DOS, and in fact argues against using undocumented DOS in TSRs.

The Waite Group's MS-DOS Developer's Guide, Second Edition, Indianapolis IN: Howard W. Sams & Co., 1989, 783 pp.
Chapter 3 ("Program and Memory Management"), chapter 4 ("Terminate and Stay Resident Programming"), chapter 6 ("Installable Device Drivers"), and chapter 11 ("Disk Layout and File Recovery") all contain discussions of undocumented DOS. There is also a brief appendix on undocumented DOS interrupts and functions.

The Waite Group's *MS-DOS Papers*, Indianapolis IN: Howard W. Sams & Co., 1988, 578 pp.
This popular book is somewhat uneven, but several chapters contain excellent discussions of undocumented DOS: chapter 6 (Raymond J. Michels, "Undocumented MS-DOS Functions," pp. 147-183), chapter 7 (M. Steven Baker, "Safe Memory-Resident Programming (TSR)," pp. 185-215), and chapter 10 (Walter Dixon, "Developing MS-DOS Device Drivers," pp. 303-344).

Richard Wilton, "Terminate-and-Stay-Resident Utilities," in Ray Duncan (ed.), *MS-DOS Encyclopedia*, pp. 347-385.
This article carries the note that "Microsoft cannot guarantee that the information in this article will be valid for future versions of MS-DOS." Nonetheless, it is probably the definitive article on TSR programming. This Microsoft-endorsed reference discusses the following undocumented DOS functions: INT 21h Function 34h (Get InDOS Flag), Function 50h (Set PSP), Function 51h (Get PSP), Function 5D0Ah (Set Extended Error Information), and INT 28h (MS-DOS Idle Interrupt). Source code in assembler.

Michael J. Young, *MS-DOS Advanced Programming*, San Francisco CA: Sybex, 1988, 490 pp.
Chapter 11 ("Memory Residency," pp. 321-360) contains an assembler module that converts a C program to a TSR, using the usual undocumented DOS functions. Chapter 12 provides a useful overview of the DOS compatibility box in OS/2.

Index

D

E

F

G

H

K

L

M

N

O

P

Q

R

S

T

U

V

W

Attention 3 1/2" disk drive users:

The *Undocumented DOS* utility disks are also available in 3 1/2" high density format. For a free exchange, please return the two 5 1/4" disks enclosed in this book to:

Undocumented DOS Disks
Trade Computer Books
Addison-Wesley Publishing Company, Inc.
1 Jacob Way
Reading, MA 01867

Be sure to enclose your name and address where you would like your disks to be sent.

IBM Programming

A ▶ BENCHMARK BOOK

> $44.95 USA
> $57.95 CANADA

Undocumented DOS

"**Undocumented DOS** *marks an important turning point for DOS programming: It collects and systematizes virtually everything that is known about DOS programming beyond the limits sanctioned by Microsoft. Only one or two books per year stand out as truly worthwhile efforts that we can use every day.* **Undocumented DOS** *is such a book. Serious DOS programmers should own a copy."*
PC Magazine

"For DOS aficionados, Addison-Wesley has a winner in **Undocumented DOS.***"*
Programmer's Journal

Since MS-DOS was first released, over 100 function calls have been listed as "reserved" by Microsoft; leaving inquiring programmers to prowl electronic bulletin boards and programming journals for information. Because such important programs as TSRs, multitasking kernels, network software, installable file systems, debuggers, Protected Mode DOS extenders, and even Windows 3.0, all make extensive use of undocumented functions, programmers must know how to use them.

Undocumented DOS puts all the power of reserved functions and data structures at your disposal in this unique book/software package. A team of expert authors provides an easy-to-follow tutorial on how to make use of undocumented functions in your programs. The first section of the book gives a general description of how to use undocumented calls, as well as warnings on when not to use them. Next are chapters on Resource Management, the DOS File System and Networks, TSRs, Multitasking, Command Interpreter, DOS Shells, and Debuggers. Then, explore MS-DOS with a debugger called INTRSPY. The authors also include a complete reference to all the undocumented functions.

Undocumented DOS contains two disks that include:

- all DOS functions — reserved and unreserved — in a hypertext pop-up utility
- the INTRSPY debugger
- DEVLOD for loading device drivers from the command line
- MEM for displaying the DOS memory chain
- ENVEDT for editing the master environment
- MONITOR and WINMON — complete assembly source code for DOS and Windows debuggers

Andrew Schulman is a software engineer at Phar Lap Software in Cambridge, MA. He is a coauthor of *Extending DOS* and a contributing editor to *Dr. Dobb's Journal.*

Raymond J. Michels is a contributor to *Programmer's Journal* and coauthored *The MS-DOS Papers.*

Jim Kyle has written extensively on MS-DOS and contributed to the *DOS Programmer's Reference* and the *MS-DOS Encyclopedia.*

Tim Paterson is the original author of MS-DOS version 1.x which he wrote from 198(Computer Products and Microsoft. He helped develop QuickBASIC and is a regular co

David Maxey manages a network software development team at a large software firm in Cambridge, MA.

Ralf Brown is a Ph.D. candidate at Carnegie Mellon University and is well-known in the on-line community for maintaining the "Interrupt List."

Cover design by Tom Tafiuri

System Requirements:
IBM or 100% compatible PC; DOS 3.0 or higher; 1.2 meg. disk drive

ISBN 0-201-57064-5
57064

Addison-Wesley Publishing Company, Inc.